The Parallel Solution of ABD Systems Arising in Numerical Methods for BVPs for ODEs

by

Richard Norman Pancer

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

The Parallel Solution of ABD Systems
Arising in Numerical Methods for BVPs for ODEs

Richard Norman Pancer
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2006

Many numerical algorithms for the solution of Boundary Value Problems (BVPs) for Ordinary Differential Equations (ODEs) contain significant components that can be parallelized easily, and recent investigations have shown that substantial speedups are attainable on machines with a modest number of processors. An important step in most of these algorithms—and often the most computationally-intensive step—is the numerical solution of an *Almost Block Diagonal* (ABD) system of linear algebraic equations. The parallelization of this step is not so straightforward as certain characteristics of the problem make it difficult to apply standard divide-and-conquer techniques in order to arrive at a stable parallel algorithm. In the past, several algorithms have been proposed for the parallel solution of the ABD system, but many are potentially unstable or offer only limited parallelism. The proper treatment of the ABD system has proven to be a challenge in the design of parallel BVP codes.

In this thesis we present three parallel algorithms for the numerical solution of ABD systems. A parallel algorithm for this problem can be up to $M/\log M$ times faster than the fastest sequential algorithm, where the fastest sequential algorithm requires M steps. Each parallel algorithm we present attains this theoretically optimal speedup if enough processors are available, and each can be adapted for use on architectures with fewer than the required number of processors. Two of the algorithms, *SLF*-QR and *SLF*-LU, were discovered independently by us and by S.J. Wright in the 1990s. Wright presented these algorithms and analyzed their stability in two papers in the 1990s, proving *SLF*-QR is stable and showing that *SLF*-LU is stable under certain assumptions. We provide some additional insight into the stability of *SLF*-LU, and

extend the basic algorithms to make better use of available processors during the factorization stage in order to increase parallelism in the solution stage.

The third algorithm we propose, RSCALE, is based on a notably different numerical technique: *eigenvalue rescaling*. RSCALE uses fewer local operations and produces less fill-in than either of the other two algorithms. In addition, RSCALE is proven to be stable for a reasonable class of problems, and has been observed to be stable for a wide class of problems through extensive numerical testing.

RSCALE is approximately 2.2 times faster than *SLF*-QR. The efficiency of *SLF*-LU is dependent on its solution strategy and its speed can vary from problem to problem, but for most problems RSCALE is approximately 1.2 times faster than *SLF*-LU. Moreover, we show that a variant of *SLF*-LU is potentially unstable on a surprising number of randomly-generated, yet well-posed, linear problems, as well as on certain nonlinear problems commonly used to test BVP codes. Since these problems pose no difficulty for either of the other two algorithms, we believe that *SLF*-QR, not *SLF*-LU, may be RSCALE's nearest competitor in terms of both speed and reliability.

Acknowledgements

I wish to thank my research supervisor, Professor Ken Jackson, for his patience and support over the years. Ken was especially helpful in providing guidance and suggesting avenues for redirection at the beginning of this project, when our original algorithms were “scooped” by another author. Thanks to the members of my thesis committee: Professors Christina Christara, Wayne Enright, Tom Fairgrieve and Rudi Mathon, and to my external examiner, Professor Patrick Keast, for reading my manuscripts and for providing insightful comments and helpful suggestions for improvement.

Special thanks to Professor Paul Muir of St. Mary’s University, Halifax, my friend and colleague. Our collaboration on some of the key topics covered in this thesis proved to be an essential ingredient in the completion of the thesis, as did Paul’s strong encouragement for me to get the job done, his relentless pursuit of me whenever I began to slip, and his invaluable assistance in proofreading many earlier drafts. Thank-you Paul.

I am grateful to the University of Toronto, and to the Department of Computer Science, for providing me the opportunity to pursue graduate studies and for their generous financial support over the years. I am also grateful to my current employer, the Department of Computer and Mathematical Sciences at the University of Toronto at Scarborough, and in particular to Professor John Scherk, the Chair of that department, for granting me a leave from teaching and for encouraging me to complete this project during my leave.

Thanks to Kirstin for her patience, moral support, and for putting up with me during the writing of this thesis, especially during the final few months. Thanks to my father, Phillip, for putting up with me for the many years before that.

Finally, I dedicate this thesis to the memory of my mother, Irene Pancer, who always believed in me and who always knew I would succeed. And I did.

Contents

List of Tables

Chapter 1

Introduction and Background

The linear systems considered in this thesis inevitably arise in the numerical solution of Boundary Value Problems (BVPs) for Ordinary Differential Equations (ODEs) of the form

$$y'(t) = f(t, y(t)), \quad t \in [t_a, t_b], \tag{1.1}$$

subject to the boundary conditions

$$g(y(t_a), y(t_b)) = 0, \tag{1.2}$$

where $y, f, g \in \mathcal{R}^n$. If the boundary conditions are separable, (1.2) can be written as

$$g(y(t_a), y(t_b)) = \begin{bmatrix} g_a(y(t_a)) \\ g_b(y(t_b)) \end{bmatrix} = 0, \tag{1.3}$$

where $g_a \in \mathcal{R}^{n_a}$ and $g_b \in \mathcal{R}^{n_b}$ with $n_a + n_b = n$.

In §1.1 a general framework for the numerical solution of (1.1) with separable or non-separable boundary conditions is outlined. Although there exists a variety of different numerical methods for solving this problem, most adhere to this general framework. Also in §1.1 the primary topic of this thesis—the Almost Block Diagonal (ABD) or Bordered Almost Block Diagonal (BABD) linear system ([Asch 88, Chapter 7])—is introduced, along with an explanation of where this system arises in the context of solving a BVP and a description of the various forms in which it can appear. There is much potential for parallelism in the numerical solution of BVPs for ODEs. This has been observed by several authors ([Benn 90], [Wrig 90], [Asch 91], [Papr 91], etc.), as has the fact that the solution to the ABD system can easily become the bottleneck in the execution time of a parallel BVP code. These issues are discussed further in §1.2. Two popular sequential codes for solving ABD systems are described in §1.3.

In §1.4 a brief history of the early attempts at developing a parallel code for solving ABD systems is given, and the first stable parallel codes are discussed in §1.5. An overview of the thesis is given in §1.6.

1.1 The Numerical Solution of BVPs for ODEs

When solving (1.1) with (1.2) or (1.3) numerically, a discrete approximation to the true continuous solution $y(t)$ is often sought. First, the interval $[t_a, t_b]$ is subdivided into a mesh:

$$t_a = t_0 < t_1 < \cdots < t_M = t_b.$$

Many techniques then proceed to compute a discrete approximation of the form

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_M \end{bmatrix} \in \mathcal{R}^{(M+1)n}$$

where $y_i \approx y(t_i)$, $y_i \in \mathcal{R}^n$, $i = 0, \ldots, M$. (We refer to y_i as the i-th *mesh variable* of the discrete approximation.) The vector Y is obtained by solving a discrete system $\Phi(Y) = 0$ that depends on the system of ODEs, the boundary conditions, and the underlying numerical method used to discretize the continuous problem. Although the residual function Φ may be written in many forms, it usually has only one component, ϕ_i, per subinterval, and often each component depends only on the unknowns local to its subinterval. If, in addition, the boundary conditions are separable, $\Phi(Y) = 0$ can be written as

$$\Phi(Y) = \begin{bmatrix} g_a(y_0) \\ \phi_0(y_0, y_1) \\ \phi_1(y_1, y_2) \\ \vdots \\ \phi_{M-1}(y_{M-1}, y_M) \\ g_b(y_M) \end{bmatrix} = 0, \tag{1.4}$$

where $\phi_i \in \mathcal{R}^n$, $i = 0, 1, \ldots, M-1$, and thus $\Phi \in \mathcal{R}^{(M+1)n}$. A variant of Newton's method is often used to solve (1.4), yielding an iteration of the form

$$\mathcal{J}^{(q)} \left[Y^{(q)} - Y^{(q+1)} \right] = \Phi(Y^{(q)}), \quad q = 0, 1, \ldots \tag{1.5}$$

where $\mathcal{J}^{(q)} \approx \partial\Phi(Y^{(q)})/\partial Y$, the Jacobian of Φ.

During each iteration, $\mathcal{J}^{(q)}$ is recomputed and the linear system (1.5) is solved. Because of the structure of Φ shown in (1.4), the Jacobian of Φ has a "staircase" or Almost Block Diagonal (ABD) form

$$\frac{\partial\Phi(Y)}{\partial Y} = \begin{bmatrix} \mathcal{B}_a & & & & & & \\ S_0 & T_0 & & & & & \\ & S_1 & T_1 & & & & \\ & & S_2 & T_2 & & & \\ & & & \ddots & \ddots & & \\ & & & & S_{M-1} & T_{M-1} \\ & & & & & \mathcal{B}_b \end{bmatrix} \in \mathcal{R}^{(M+1)n\times(M+1)n} \tag{1.6}$$

where

$$S_i = \frac{\partial\phi_i(y_i, y_{i+1})}{\partial y_i} \in \mathcal{R}^{n\times n}, \quad T_i = \frac{\partial\phi_i(y_i, y_{i+1})}{\partial y_{i+1}} \in \mathcal{R}^{n\times n}, \quad i = 0, 1, \ldots, M-1,$$

and

$$\mathcal{B}_a = \frac{\partial g_a(y_0)}{\partial y_0} \in \mathcal{R}^{n_a\times n}, \quad \mathcal{B}_b = \frac{\partial g_b(y_M)}{\partial y_M} \in \mathcal{R}^{n_b\times n}.$$

The main objective of this thesis is to develop parallel algorithms for the numerical solution of ABD linear systems.

Each iteration of (1.5) requires the evaluation and factorization of $\mathcal{J}^{(q)}$, the evaluation of $\Phi(Y^{(q)})$, and a forward elimination and back-solve. In practice, $\mathcal{J}^{(q)}$ may be held fixed for several iterations provided an acceptable rate of convergence is achieved. In this case, assuming the factors of $\mathcal{J}^{(q)}$ are stored, several linear systems may be solved at a substantially reduced cost with only a forward elimination and back-solve on subsequent right-hand sides $\Phi(Y^{(k)})$, $k = q+1, q+2, \ldots$.

If the BVP is posed with non-separable boundary conditions, $\Phi(Y) = 0$ may be written as

$$\Phi(Y) = \begin{bmatrix} g(y_0, y_M) \\ \phi_0(y_0, y_1) \\ \phi_1(y_1, y_2) \\ \vdots \\ \phi_{M-1}(y_{M-1}, y_M) \end{bmatrix} = 0, \tag{1.7}$$

where $g(y_0, y_M) \in \mathcal{R}^n$ and ϕ_i, $i = 0, 1, \ldots, M-1$ are as defined in (1.4). In this case, the Jacobian of Φ has a slightly modified form. Augmented with its right-hand side, the linear

system that must be solved at each iteration of (1.5) appears as

$$[\mathcal{J}_2|\Phi] = \left[\begin{array}{ccccccc|c} \mathcal{B}_a & & & & & & \mathcal{B}_b & g \\ S_0 & T_0 & & & & & & \phi_0 \\ & S_1 & T_1 & & & & & \phi_1 \\ & & S_2 & T_2 & & & & \phi_2 \\ & & & \ddots & \ddots & & & \vdots \\ & & & & S_{M-1} & T_{M-1} & & \phi_{M-1} \end{array}\right] \tag{1.8}$$

where $[S_i, T_i],\ i = 0, 1, \ldots, M-1$ are as defined in (1.6), and

$$\mathcal{B}_a = \frac{\partial g(y_0, y_M)}{\partial y_0} \in \mathcal{R}^{n\times n},\quad \mathcal{B}_b = \frac{\partial g(y_0, y_M)}{\partial y_M} \in \mathcal{R}^{n\times n}.$$

Although $\mathcal{J}_2$ is not, by precise definition, an ABD matrix, it is referred to as such throughout this thesis so as to avoid having to maintain a distinction between the two Jacobian structures. This second form of Jacobian—often called a Bordered Almost Block Diagonal (BABD) matrix—normally will be used when describing parallel algorithms for solving ABD systems since none of the new algorithms we present require separable boundary conditions. Note that if the BVP *is* posed with separable boundary conditions, (1.6) can be transformed easily to (1.8) using an appropriate row permutation.

Finally, some algorithms described later are slightly more efficient in terms of both storage and speed when $T_i = I,\ i = 0, \ldots, M-1$:

$$[\mathcal{J}_3|\Phi] = \left[\begin{array}{ccccccc|c} \mathcal{B}_a & & & & & & \mathcal{B}_b & g \\ V_0 & I & & & & & & \phi_0 \\ & V_1 & I & & & & & \phi_1 \\ & & V_2 & I & & & & \phi_2 \\ & & & \ddots & \ddots & & & \vdots \\ & & & & V_{M-1} & I & & \phi_{M-1} \end{array}\right] \tag{1.9}$$

This third form of Jacobian arises when

$$\phi_i(y_i, y_{i+1}) \equiv y_{i+1} + \sigma_i(y_i),\ i = 0, 1, \ldots, M-1 \tag{1.10}$$

in (1.7), with $\sigma_i \in \mathcal{R}^n$ depending only on y_i. This form of residual occurs, for example, in multiple shooting and codes based on implicit Runge-Kutta formulas (see [Asch 88]).

There are several other issues that need be addressed in the numerical solution of BVPs, including selecting an appropriate mesh, choosing a specific formula for the residual function, and determining a convergence criteria for (1.5). See [Asch 88] for details.

1.2 The Potential for Parallelism in a BVP Code

Much parallelism is inherent and obvious in the approach outlined in §1.1. For example, the residual components of $\Phi(Y^{(q)})$ can be evaluated independently, and the block-pairs $[S_i, T_i]$ of $\mathcal{J}^{(q)}$ can be constructed independently each time $\mathcal{J}^{(q)}$ is re-evaluated in (1.5). In [Benn 90], a parallel version of COLNEW [Bade 87] is implemented and the speed-up achievable by parallelizing the Jacobian set-up phase is investigated. A high percentage (60-80%) of the total execution time in COLNEW is spent during set-up where large blocks are "condensed" to form (1.6). Condensing in parallel, therefore, is very effective and in fact shifts the most time-consuming phase of the code: run-time profiling in [Benn 90] shows that the factorization and solution of the condensed ABD system becomes the bottleneck in the parallel implementation. In codes that do not use condensation the resulting ABD system can be much larger, and its factorization and solution an even bigger bottleneck. In COLSYS [Asch 81], for example, the ABD factorization often accounts for more than 50% of the total execution time [Muir 91]. These statistics emphasize the importance of developing a parallel ABD system solver.

A detailed history of the development of parallel software for BVPs for ODEs (BVODEs) is given in [Muir 03]. The focus of that paper is the parallel BVODE code PMirkDC which incorporates, as its parallel ABD system solver, the RSCALE algorithm presented in this thesis. Further experiments with PMirkDC are included in Chapter 4 of this thesis.

1.3 Sequential Codes for Solving ABD Systems

Efficient sequential codes for solving ABD systems have been available for several years—two examples are SOLVEBLOK [deBo 80] and COLROW [Diaz 83]. Both codes perform the factorization in $\mathcal{O}(Mn^3)$ time. SOLVEBLOK eliminates by rows and COLROW uses a variation of alternate row and column elimination to avoid fill-in. Stability is achieved by a pivoting strategy which controls element growth during the factorization. The pivoting results in an implicit decoupling of the underlying increasing and decreasing fundamental solution modes, an essential process for the stable solution of a BVP ([Asch 88, Chapter 6]).

The potential for parallelism in these codes is *not* obvious. Most are essentially variations of Gaussian elimination with row partial-pivoting, and hence are inherently sequential. It is sometimes possible, however, to first partition (1.6) and then factor each partition independently using one of these sequential codes. This approach has met with limited success, and is discussed further in the next section.

1.4 Early Attempts at Parallel Codes

The greatest potential for speed-up in the parallel solution of ABD linear systems exists across the blocks (i.e. in M), since typically $M \gg n$ in a BVP. Parallelism within the blocks is complementary, and could be exploited also if n is sufficiently large, but this possibility is not pursued here. Suppressing the dependence on n, therefore, the theoretically best parallel complexity is $\mathcal{O}(\log M)$ block-steps[1] since block-rows must be processed pairwise during the factorization. Most parallel algorithms proposed in the past, however, either do not achieve the optimal speed-up or suffer from poor stability properties. Following is a brief history of some of the more noteworthy contributions:

1989 Lentini [Lent 89] suggests performing Gaussian elimination with pivoting simultaneously from both ends of the matrix. This leads to a stable algorithm that effectively uses two processors, but it cannot be generalized for greater parallelism.

1990 Wright and Pereyra [Wrig 90] present a block factorization algorithm which is essentially equivalent to *compactification*—an algorithm known to be potentially unstable ([Asch 88, page 153]). They propose using the parallel algorithm initially, and when instability is detected, switching to a more stable sequential method.

1991 Paprzycki and Gladwell [Papr 91] describe a "tearing" algorithm in which the original ABD matrix is partitioned into several smaller matrices, each of which is ABD. These represent sub-BVPs which can be solved independently using an existing sequential code, after which the solutions are combined. Unfortunately, although it is always possible to select intermediate boundary conditions and construct the smaller ABD systems, there is no guarantee that the sub-problems will be well-conditioned. In addition, the authors found that even when a problem could be solved stably with this method, the speed-up achieved was less than expected.

1991 Ascher and Chan [Asch 91] propose solving the ABD system by first forming the normal equations. The resulting system is symmetric, positive-definite, block-tridiagonal and can be solved stably in $\mathcal{O}(\log M)$ block-steps using a variation of block cyclic reduction. (See [Hell 76]; we also describe the block cyclic reduction algorithm in this thesis.) The drawback here, of course, is that by forming the normal equations explicitly the

[1] $\log M \equiv \log_2 M$.

2-norm condition number of the original system is squared. Also, a block of fill-in is introduced in each block-row which potentially leads to higher operation counts.

1.5 The First Stable Parallel Code and Thesis Goals

Wright was the first to publish a *stable* parallel algorithm that attains the theoretically optimal speed-up for this problem. In [Wrig 92] a "structured orthogonal factorization" is described which—when embedded in a cyclic reduction algorithm—solves the ABD system stably in $\mathcal{O}(\log M)$ block-steps with as few as $M/\log M$ processors. Stability is assured since the algorithm (Structured QR) is essentially a QR-factorization applied to a row and column permuted version of the original ABD matrix. In [Wrig 94], Wright replaces the local orthogonal transformations used in [Wrig 92] with Gauss transformations with row partial-pivoting resulting in a substantial speed-up even though the number of block-steps is the same. The algorithm (Structured LU) is equivalent to Gaussian elimination with restricted row partial-pivoting applied to the original ABD matrix, and is stable for a wide range of problems, although rapid error growth can arise in some cases [Wrig 93].

Since Wright published his Structured QR and Structured LU papers, several authors have proposed minor modifications to the basic algorithms to help improve speedup and/or stability. The local operation count per block-step, however, remains the same in all variations. A review of recent parallel (and sequential) solution techniques for solving ABD linear systems is given in [Amod 00].

The algorithms *SLF*-QR and *SLF*-LU described in §2.2 of this thesis were discovered independently by us [Panc 92], and are similar to those of Wright. Unfortunately, before we could complete our analysis of the algorithms, Wright published Structured QR along with a proof of its stability. Once this happened, the focus of the thesis changed somewhat from our original goal of discovering the *first* optimally parallel stable algorithm, and the following additional goals were set:

- Extend *SLF*-QR and *SLF*-LU to increase parallelism to $\mathcal{O}(1)$ block-steps in the back-solve stage (i.e., constant complexity, independent of M) by making better use of available processors during the decomposition stage.

- Further analyze the potential for instability in *SLF*-LU, which was initially addressed in [Wrig 93] and [Wrig 94], and attempt to measure the reliability of this algorithm when used in a production code for solving nonlinear BVPs for ODEs.

- Develop a new algorithm—RSCALE—based on a different numerical technique, which does not exhibit the instability inherent in *SLF*-LU, and which is significantly faster than *SLF*-QR due to reduced local operation counts.

- Implement a robust parallel FORTRAN code for each of *SLF*-QR, *SLF*-LU and RSCALE using state-of-the-art mathematical software (level-3 BLAS), show that each code speeds up linearly with the number of processors on a parallel machine, and show that the relative execution times of the three codes agree with what one would expect from the local operation counts.

- Assess the relative performance of the three codes, in terms of both accuracy and speed, when they are incorporated in MirkDC [Enri 96], a software package for solving nonlinear BVPs for ODEs.

These additional goals were met by the completion of the thesis.

1.6 Overview of the Thesis

An overview of the remainder of the thesis is given below.

Chapter 2: Several variations of *SLF*-QR, *SLF*-LU and RSCALE are described. Each algorithm is presented as a modified form of *block cyclic reduction* (BlkCR), a generalization of the well-known cyclic reduction algorithm for solving tridiagonal linear systems.

Chapter 3: The stability of the three algorithms is discussed. A thorough stability analysis of *SLF*-QR and *SLF*-LU is given in [Wrig 92] and [Wrig 94]—only the key points are reviewed here. However, the potential for instability in *SLF*-LU, which was initially addressed in [Wrig 93] and [Wrig 94], is investigated in greater depth. The chapter closes with a stability analysis of RSCALE, applicable when the algorithm is used to solve a certain class of ABD system.

Chapter 4: The three algorithms are compared with respect to ease of implementation, storage requirements, accuracy and speed. Detailed operation counts are derived for the local operations performed at each block step. Numerical tests run on a sequential machine show that the relative execution times of the three algorithms agree well with what one would expect from the local operation counts. Numerical tests run on a parallel machine show that all three algorithms exhibit close to linear speedup when used to solve

sufficiently large ABD systems. The relative performance of the algorithms is assessed when the codes are incorporated in MirkDC [Enri 96], a software package for solving nonlinear BVPs for ODEs.

Chapter 5: The thesis concludes with a summary and discussion of future work.

The appendices contain:

- additional RSCALE stability results from the analysis in Chapter 3,
- additional experiments demonstrating the relative performance of the three solvers on a sequential machine,
- additional experiments demonstrating the relative performance of the three solvers on a parallel machine,
- additional experiments with the solvers incorporated in MirkDC, and
- a complete source listing of the *SLF*-QR, *SLF*-LU and RSCALE parallel codes.

Chapter 2

Description of the Algorithms

In this chapter we describe several variations of three parallel algorithms for solving ABD linear systems. Each variation is based on a generalization of *cyclic reduction*, an algorithm originally proposed for solving tridiagonal linear systems. In §2.1, the basic (and unstable) cyclic reduction algorithm is outlined. In §2.2 the block multiplications used in the basic algorithm are replaced by *Stable Local Factorization* (*SLF*) transformations. These transformations give rise to two similar parallel algorithms for solving the ABD system: *SLF*-QR and *SLF*-LU. In §2.3 a third algorithm based on a different numerical technique is presented. RSCALE uses *eigenvalue rescaling* to first transform the ABD matrix so that block cyclic reduction as described in §2.1 can be performed stably. Although the number of block-steps is the same as that of the algorithms in §2.2, RSCALE requires fewer local operations within each block-step and therefore potentially is faster than either of the SLF-based algorithms. We provide a detailed complexity analysis of the algorithms later in the thesis.

2.1 Block Cyclic Reduction

Cyclic reduction was originally proposed as a stable sequential algorithm for solving symmetric positive-definite tridiagonal linear systems ([Hock 65], [Hock 70], [Golu 83]). It does, however, possess considerable inherent parallelism and can be readily adapted to solve ABD systems. To this end, consider the third form of the ABD system (1.9). If this form does not arise naturally from the discretization, it can be computed by multiplying the i-th block-row of (1.8) through by T_i^{-1} (providing T_i^{-1} exists and is not too badly conditioned)—a process which is obviously highly parallel. Now assuming $M/2$ processors are available initially, the first step or "sweep" of the reduction assigns each pair of block-rows $2i$ and $2i+1$ to a proces-

sor, multiplies row i by V_{i+1} and subtracts it from row $i+1$, giving[1]

$$\left[\begin{array}{cccccccc|c}
\mathcal{B}_a & & & & & & & \mathcal{B}_b & g \\
V_0 & I & & & & & & & \phi_0 \\
-V_1V_0 & & I & & & & & & \phi_1 - V_1\phi_0 \\
 & & V_2 & I & & & & & \phi_2 \\
 & & -V_3V_2 & & I & & & & \phi_3 - V_3\phi_2 \\
 & & & & \ddots & & \ddots & & \vdots \\
 & & & & & & V_{M-2} & I & \phi_{M-2} \\
 & & & & & -V_{M-1}V_{M-2} & & I & \phi_{M-1} - V_{M-1}\phi_{M-2}
\end{array}\right] \tag{2.1}$$

This effectively uncouples the odd-indexed unknowns from (1.9). Once $y_{2i}, i = 0 \ldots M/2$, have been computed, $y_{2i+1}, i = 0 \ldots M/2 - 1$, can be obtained from (2.1) by

$$y_{2i+1} = \phi_{2i} - V_{2i}y_{2i}, \quad i = 0, \ldots, M/2 - 1. \tag{2.2}$$

The even-indexed unknowns are computed during the second sweep by solving a reduced or compacted ABD system constructed from the odd rows and columns of (2.1):

$$\left[\begin{array}{cccccc|c}
\mathcal{B}_a & & & & & \mathcal{B}_b & g \\
-V_1V_0 & I & & & & & \phi_1 - V_1\phi_0 \\
 & -V_3V_2 & I & & & & \phi_3 - V_3\phi_2 \\
 & & \ddots & & \ddots & & \vdots \\
 & & & -V_{M-1}V_{M-2} & I & & \phi_{M-1} - V_{M-1}\phi_{M-2}
\end{array}\right] \tag{2.3}$$

The algorithm proceeds recursively for $\log M$ sweeps—each sweep utilizing half the processors of the previous one—resulting in a final $2n \times 2n$ compacted system. This small system is solved sequentially by a stable method, such as Gaussian elimination, to obtain y_0 and y_M, and then $\log M$ back-solve sweeps of the form (2.2) are used to recover the remaining unknowns. During the back-solve, the number of active processors doubles with each sweep.

Other variants of this algorithm are possible. The need for $\log M$ back-solve sweeps can be eliminated by continuously using $M/2$ processors during the reduction. This results in a

[1] Block-elements of the ABD matrix appearing as white-space, such as the element between $-V_1V_0$ and I in block-row 1 of (2.1), are understood to be the $n \times n$ zero matrix. This convention is used throughout the thesis.

system of the form

$$\left[\begin{array}{ccccc|c} \mathcal{B}_a & & & & \mathcal{B}_b & g \\ V_0 & I & & & & \tilde{\phi}_0 \\ -V_1V_0 & & I & & & \tilde{\phi}_1 \\ V_2V_1V_0 & & & I & & \tilde{\phi}_2 \\ -V_3V_2V_1V_0 & & & & I & \tilde{\phi}_3 \\ \vdots & & & \ddots & & \vdots \\ (-1)^{M-1}\prod_{i=0}^{M-1} V_{M-1-i} & & & & I & \tilde{\phi}_{M-1} \end{array}\right] \tag{2.4}$$

where $\tilde{\phi}_0 = \phi_0, \tilde{\phi}_i = \phi_i - V_i\tilde{\phi}_{i-1}, i = 1 \ldots M-1$. Then y_0 and y_M are computed by solving

$$\left[\begin{array}{cc} \mathcal{B}_a & \mathcal{B}_b \\ (-1)^{M-1}\prod_{i=0}^{M-1} V_{M-1-i} & I \end{array}\right] \left[\begin{array}{c} y_0 \\ y_M \end{array}\right] = \left[\begin{array}{c} g \\ \tilde{\phi}_{M-1} \end{array}\right] \tag{2.5}$$

the same $2n \times 2n$ compacted system that arises in the first version. The back-solve for (2.4) is completely parallel: the remaining unknowns can be computed in 1 step on M processors or 2 steps on $M/2$ processors.

If only $P < M/2$ processors are available, each processor can be assigned $M/2P$ pairs of block-rows to process in the first sweep, with the number of pairs of block-rows decreasing by a factor of 2 with each sweep until there are more processors than pairs of block-rows. Alternatively, the system could first be partitioned into P blocks of M/P block-rows. Each partition is assigned to a single processor, where it is reduced using a sequential algorithm. This results in a compacted system of order P that can be solved using cyclic reduction as shown above. Using this technique, $\mathcal{O}(\log M)$ complexity is attainable with as few as $P = M/\log M$ processors. See [Asch 91] for more details.

Unfortunately, none of these algorithms is appropriate for solving ABD systems arising from numerical methods for BVPs. As pointed out in [Asch 91], they are equivalent to compactification—an algorithm that is known to be unstable because it fails to decouple the fast increasing and decreasing fundamental solution modes. Fundamental solution modes are characteristic of the underlying *dichotomy* of the differential equation. Well posed BVPs exhibit such a dichotomy. The increasing solution modes are controlled by the right boundary conditions and the decreasing solution modes are controlled by the left boundary conditions. These concepts are defined more formally in [Asch 88, Chapter 6], and in Chapter 3 of this thesis where we discuss the stability of the algorithms.

In the following sections, we propose several modifications to the basic cyclic reduction algorithm in order to improve its stability when used to solve the ABD system.

2.2 Stable Local Factorization

The cyclic reduction algorithm described in §2.1 uses the elementary transformation

$$\left[\begin{array}{cccc|c} \cdots & V_i & I & \cdots & \phi_i \\ \cdots & -V_{i+1}V_i & I & \cdots & \phi_{i+1} - V_{i+1}\phi_i \end{array}\right] \longleftarrow \left[\begin{array}{cc} I & 0 \\ -V_{i+1} & I \end{array}\right] \times \left[\begin{array}{cccc|c} \cdots & V_i & I & \cdots & \phi_i \\ \cdots & & V_{i+1} & I & \cdots & \phi_{i+1} \end{array}\right] \tag{2.6}$$

The algorithm is potentially unstable if, for example, some eigenvalues of V_i and/or V_{i+1} are greater than one in magnitude, which is typical of ABD matrices arising from the discretization of BVPs—the larger eigenvalues correspond to increasing fundamental solution modes. As the reduction progresses, $\| \prod V_i \|$ can grow rapidly resulting in the loss of all significant digits to machine round-off.

In this section, we propose two alternative transformations which can be adapted for use in a variation of the cyclic reduction algorithm. These *Stable Local Factorization* (SLF) transformations give rise to stable parallel algorithms for solving ABD systems. Each avoids the instability inherent in the block multiplications of (2.6) by controlling the growth of elements during the reduction. One uses orthogonal factorization, the other Gaussian elimination with row partial-pivoting. The stability of the new algorithms is analyzed in [Wrig 92], [Wrig 94] and Chapter 3 of this thesis. (As is shown in [Wrig 94] and §3.2, the prefix "SLF" is perhaps a bit of a misnomer for the Gaussian elimination variant, as this variant can be unstable on certain classes of problems. Nevertheless, we find it convenient to label the orthogonal factorization and Gaussian elimination variants of the new algorithm with the same prefix in order to be consistent with Wright's presentation, and also because they are structurally equivalent.)

Since SLF-transformations do not exploit the identity matrix, the second form of the ABD system (1.8) is used throughout this section; the i-th slice appears as

$$\left[\begin{array}{ccccc|c} \cdots & S_i & T_i & & \cdots & \phi_i \\ \cdots & & S_{i+1} & T_{i+1} & \cdots & \phi_{i+1} \end{array}\right]$$

The notation $S_i^{(k)}$ is used to indicate that the matrix S_i has been transformed during the k-th step of an algorithm.

2.2.1 Orthogonal Factorization

This transformation requires the QR-factorization of part of the i-th slice:

$$\left[\begin{array}{c} T_i \\ S_{i+1} \end{array}\right] = Q_i \left[\begin{array}{c} R_i \\ 0 \end{array}\right]$$

where $Q_i \in \mathcal{R}^{2n\times 2n}$ is orthogonal and $R_i \in \mathcal{R}^{n\times n}$ is upper-triangular. Conceptually, once Q_i is obtained the i-th slice is multiplied through by Q_i^T giving

$$\left[\begin{array}{ccccc|c} \cdots & S_i^{(k)} & R_i^{(k)} & T_i^{(k)} & \cdots & \phi_i^{(k)} \\ \cdots & S_{i+1}^{(k)} & & T_{i+1}^{(k)} & \cdots & \phi_{i+1}^{(k)} \end{array}\right] \longleftarrow Q_i^T \times \left[\begin{array}{ccccc|c} \cdots & S_i & T_i & & \cdots & \phi_i \\ \cdots & & S_{i+1} & T_{i+1} & \cdots & \phi_{i+1} \end{array}\right] \tag{2.7}$$

However, in practice, the QR-factorization is computed in place with Householder reflections which are stored in the lower triangle of $R_i^{(k)}$ and in the space formerly occupied by the elements in the $n \times n$ block immediately below $R_i^{(k)}$. Thus, in the context of a modified Newton iteration (1.5), it is possible to perform several iterations (i.e. transform several subsequent right-hand sides) without recomputing the QR-factorization.

The *SLF*-QR transformation is structurally different than (2.6); in particular, there is fill-in at $T_i^{(k)}$. Therefore, the algorithms in §2.1 must be modified for use with *SLF*-QR. This is discussed further in §2.2.3 and §2.2.4.

2.2.2 Gaussian Elimination with Row Partial-Pivoting

This transformation is similar to *SLF*-QR, but instead uses an LU-factorization:

$$\left[\begin{array}{c} T_i \\ S_{i+1} \end{array}\right] = L_i \left[\begin{array}{c} U_i \\ 0 \end{array}\right]$$

where $L_i^{-1} = \tilde{L}_n P_n \cdots \tilde{L}_2 P_2 \tilde{L}_1 P_1 \in \mathcal{R}^{2n\times 2n}$, $\tilde{L}_j$ is an elementary Gauss transformation, P_j is a permutation matrix, and $U_i \in \mathcal{R}^{n\times n}$ is upper-triangular. Conceptually, once L_i^{-1} is obtained the i-th slice is multiplied through by L_i^{-1} giving

$$\left[\begin{array}{ccccc|c} \cdots & \tilde{S}_i^{(k)} & U_i^{(k)} & \tilde{T}_i^{(k)} & \cdots & \tilde{\phi}_i^{(k)} \\ \cdots & \tilde{S}_{i+1}^{(k)} & & \tilde{T}_{i+1}^{(k)} & \cdots & \tilde{\phi}_{i+1}^{(k)} \end{array}\right] \longleftarrow L_i^{-1} \times \left[\begin{array}{ccccc|c} \cdots & S_i & T_i & & \cdots & \phi_i \\ \cdots & & S_{i+1} & T_{i+1} & \cdots & \phi_{i+1} \end{array}\right] \tag{2.8}$$

However, in practice, the factorization is computed in place with Gauss transformations which are stored in the space formerly occupied by the elements below $U_i^{(k)}$, and the permutation matrices are stored in a single integer vector of length n. As with *SLF*-QR, several modified Newton iterations (1.5) can be computed using the same *SLF*-LU factorization.

SLF-LU is obviously structurally equivalent to *SLF*-QR. Either can be used with the algorithms in §2.2.3 and §2.2.4.

2.2.3 The SLF Partitioning Algorithm

SLF-transformations can be incorporated in a partitioning algorithm which generalizes cyclic reduction for use on architectures with $P \leq M$ processors. The ABD system is first partitioned

into P blocks of M/P block-rows each; the first partition appears as

$$\left[\begin{array}{cccccc|c} S_0 & T_0 & & & & & \phi_0 \\ & S_1 & T_1 & & & & \phi_1 \\ & & S_2 & T_2 & & & \phi_2 \\ & & & S_3 & T_3 & & \phi_3 \\ & & & & \ddots & \ddots & \vdots \\ & & & & & S_{m_0} \quad T_{m_0} & \phi_{m_0} \end{array}\right] \tag{2.9}$$

where $m_0 + 1$ is the number of block-rows in the first partition. The reduction steps shown below for the first partition can be carried out simultaneously on all P partitions. The reduction starts at the top and works down. In the first step a factorization is computed for $[T_0^T S_1^T]^T$ and (2.9) is transformed to

$$\left[\begin{array}{cccccc|c} \boxed{S_0^{(1)}} & \boxed{R_0^{(1)}} & \boxed{T_0^{(1)}} & & & & \boxed{\phi_0^{(1)}} \\ \boxed{S_1^{(1)}} & & \boxed{T_1^{(1)}} & & & & \boxed{\phi_1^{(1)}} \\ & & S_2 & T_2 & & & \phi_2 \\ & & & S_3 & T_3 & & \phi_3 \\ & & & & \ddots & \ddots & \vdots \\ & & & & & S_{m_0} \quad T_{m_0} & \phi_{m_0} \end{array}\right] \tag{2.10}$$

In the second step $[T_1^{(1)T} S_2^T]^T$ is factored and (2.10) is transformed to

$$\left[\begin{array}{cccccc|c} S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & \phi_0^{(1)} \\ \boxed{S_1^{(2)}} & & \boxed{R_1^{(2)}} & \boxed{T_1^{(2)}} & & & \boxed{\phi_1^{(2)}} \\ \boxed{S_2^{(2)}} & & & \boxed{T_2^{(2)}} & & & \boxed{\phi_2^{(2)}} \\ & & & S_3 & T_3 & & \phi_3 \\ & & & & \ddots & \ddots & \vdots \\ & & & & & S_{m_0} \quad T_{m_0} & \phi_{m_0} \end{array}\right] \tag{2.11}$$

After m_0 steps the partition has the form

$$\left[\begin{array}{cccccc|c} S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & \phi_0^{(1)} \\ S_1^{(2)} & & R_1^{(2)} & T_1^{(2)} & & & \phi_1^{(2)} \\ S_2^{(3)} & & & R_2^{(3)} & T_2^{(3)} & & \phi_2^{(3)} \\ \vdots & & & & \ddots & \ddots & \vdots \\ \boxed{S_{m_0-1}^{(m_0)}} & & & & & \boxed{R_{m_0-1}^{(m_0)}} \quad \boxed{T_{m_0-1}^{(m_0)}} & \boxed{\phi_{m_0-1}^{(m_0)}} \\ \boxed{S_{m_0}^{(m_0)}} & & & & & \boxed{T_{m_0}^{(m_0)}} & \boxed{\phi_{m_0}^{(m_0)}} \end{array}\right] \tag{2.12}$$

Once y_0 and y_{m_0+1}, the end mesh variables of this partition, are known, the interior mesh variables are computed by back-substitution; each step requires solving the triangular system

$$R_{i-1}^{(i)} y_i = \phi_{i-1}^{(i)} - S_{i-1}^{(i)} y_0 - T_{i-1}^{(i)} y_{i+1}, \quad i = m_0, m_0 - 1, \ldots, 1. \tag{2.13}$$

As with the reduction, this can be done simultaneously on all P partitions.

The end mesh variables for each partition are computed by solving a compacted system of order P constructed from the last block-row of each partition. Letting $[S_{m_i}^{(m_i)}, T_{m_i}^{(m_i)}, \phi_{m_i}^{(m_i)}] \equiv [S_{p_i}, T_{p_i}, \phi_{p_i}], i = 0 \ldots P-1$, this compacted system has the form

$$\left[\begin{array}{ccccccc|c} \mathcal{B}_a & & & & & & \mathcal{B}_b & g \\ S_{p_0} & T_{p_0} & & & & & & \phi_{p_0} \\ & S_{p_1} & T_{p_1} & & & & & \phi_{p_1} \\ & & S_{p_2} & T_{p_2} & & & & \phi_{p_2} \\ & & & \ddots & \ddots & & & \vdots \\ & & & & S_{p_{P-1}} & T_{p_{P-1}} & & \phi_{p_{P-1}} \end{array}\right] \tag{2.14}$$

which has the same structure as (1.8). It could therefore be solved in $\mathcal{O}(P)$ block-steps using a stable sequential code, but since P processors are available it is possible to apply cyclic reduction to solve it in $\mathcal{O}(\log P)$ block-steps. Two versions of cyclic reduction using SLF-transformations are described below. To simplify the figures, P is set to 8 and the right-hand sides are omitted. Generalizing the algorithms for more processors is straightforward.

The first version (Figure 2.1) requires $\log P$ reduction sweeps and $\log P$ back-solve sweeps. During the first reduction sweep 4 processors are active computing SLF-transformations simultaneously on block-rows 0-1, 2-3, 4-5, and 6-7. During the second sweep, 2 processors are active transforming block-rows 1-3 and 5-7, and during the final sweep 1 processor is active transforming block-rows 3-7. y_{p_0} and y_{p_7+1} are then computed by solving a $2n \times 2n$ system constructed from the first and last block-rows:

$$\left[\begin{array}{cc} \mathcal{B}_a & \mathcal{B}_b \\ S_{p_7}^{(3)} & T_{p_7}^{(3)} \end{array}\right] \left[\begin{array}{c} y_{p_0} \\ y_{p_7+1} \end{array}\right] = \left[\begin{array}{c} g \\ \phi_{p_7}^{(3)} \end{array}\right] \tag{2.15}$$

The back-solve works in a reverse fashion. During the first sweep 1 processor is active solving the triangular system in block-row 3 for y_{p_4}. During the second sweep 2 processors are active solving systems in block-rows 1 and 5 for y_{p_2} and y_{p_6}, and during the final sweep 4 processors are active solving systems in block-rows 0, 2, 4, and 6 for y_{p_1}, y_{p_3}, y_{p_5}, and y_{p_7}.

Several subsequent right-hand sides can be transformed during a modified Newton iteration (1.5) without recomputing the local factorizations. If the Householder or Gauss transformations

Figure 2.1: Cyclic reduction using stable local factorization

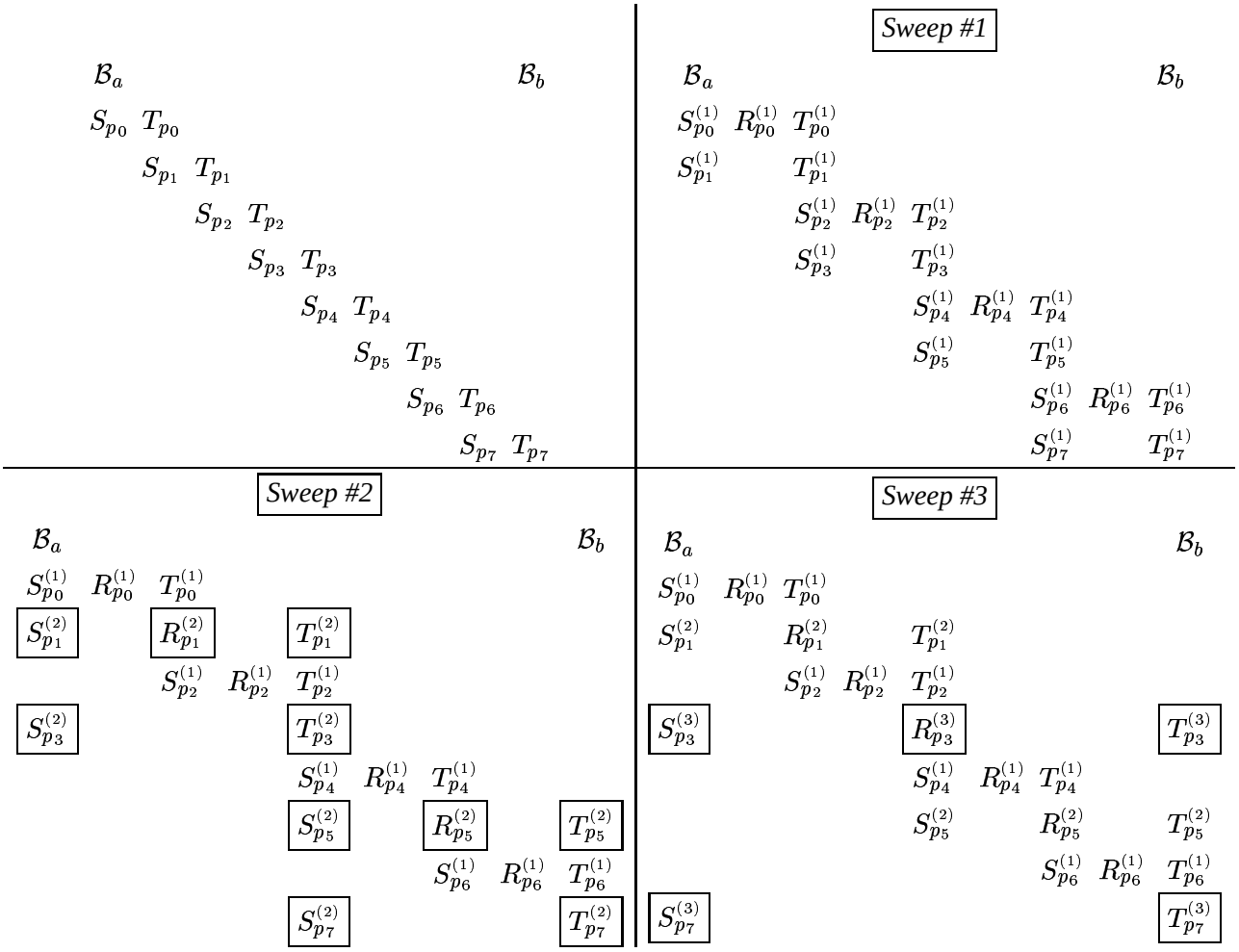
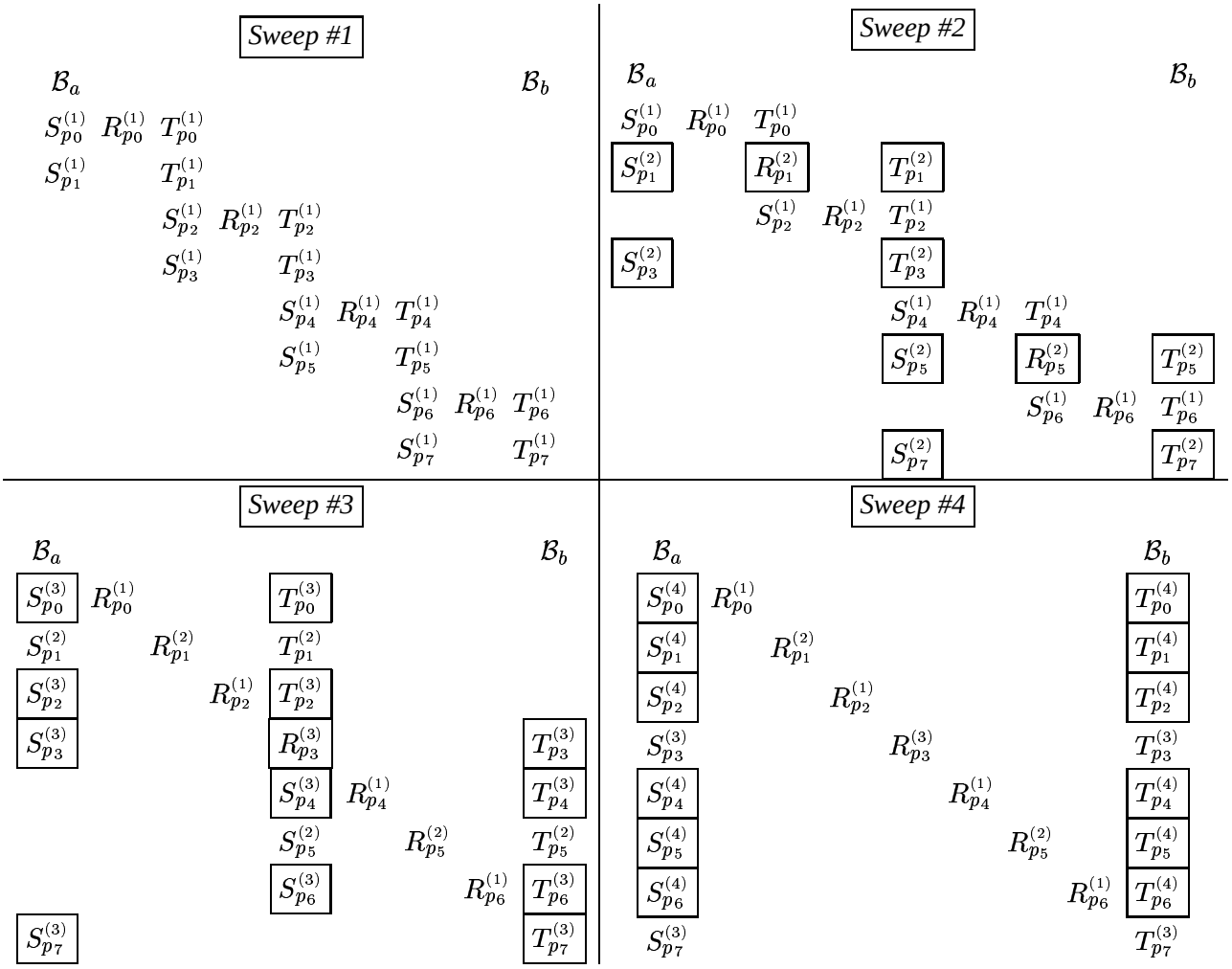


Figure 2.2: Cyclic reduction to N-shape Jacobian using stable local factorization



are stored as explained in §2.2.1 or §2.2.2, the reduction sweeps simply involve applying the stored transformations to the new right-hand side at a cost per block-row pair of $3n^2$ flops[2] for a Householder transformation or $\frac{3}{2}n^2$ flops for a Gauss transformation.

In the second version (Figure 2.2) the need for $\log P$ back-solve sweeps is eliminated by making better use of processors during the reduction. Processors that were idle in Figure 2.1 are now used to "push" S and T blocks outwards resulting in an *N*-shaped Jacobian. Clearly, once y_{p_0} and y_{p_7+1} are known, the back-solve on this Jacobian can be accomplished in 1 sweep using P processors. The push begins at sweep #3 and requires an additional sweep at the end. The savings of a completely parallel back-solve, however, can easily outweigh the overhead of

[2]We define a *flop*, or *floating-point operation*, formally in §4.1.

the additional reduction sweep when the Jacobian is kept fixed for several iterations in (1.5).

The usual SLF-transformation keeps $P/2^k$ processors active during sweep k. The S and T block pushes keep an additional $P - 4P/2^k$ processors active during sweeps $k = 3, \ldots, 1 + \log P$. Therefore, for this version,

$$\text{total \# of active processors} = \begin{cases} P/2^k & 1 \le k \le 2 \\ P - 3P/2^k & 3 \le k \le \log P \\ P - 2 & k = 1 + \log P. \end{cases}$$

Thus, in contrast to the first version, the number of active processors *increases* during the reduction to a maximum of $P - 2$.

In order for the second version to be efficient each processor must push its blocks in the same time required for the SLF-transformation (or less). For example, in sweep #3 of Figure 2.2 one processor computes the usual SLF-transformation on block-rows 3-7. Simultaneously, 4 other processors push S and T blocks outwards in rows 0, 2, 4, and 6. As shown below for block-row 0, a "push" requires 4 steps:

1. $V = (R_{p_1}^{(2)})^{-1} S_{p_1}^{(2)}$

2. $W = (R_{p_1}^{(2)})^{-1} T_{p_1}^{(2)}$

3. $S_{p_0}^{(3)} = S_{p_0}^{(1)} - T_{p_0}^{(1)} V$

4. $T_{p_0}^{(3)} = -T_{p_0}^{(1)} W$

Since $R_{p_1}^{(2)}$ is upper-triangular, steps 1 and 2 require $n^3/2$ flops each. The total for all four steps is therefore $3n^3 + \mathcal{O}(n^2)$ which, fortunately, is less than the flop count for the fastest SLF-transformation (see Chapter 4).

One possible way of optimizing this process stems from the observation that often in a BVP the boundary conditions have little influence on the solution at interior mesh points. As a result, in many problems the interior S and T blocks "vanish" long before they reach their respective left and right boundaries. Numerical experiments show that these blocks can be dropped when their norm falls below a threshold tolerance (close to machine epsilon) without affecting the solution. Of course the overhead of monitoring the norm must be taken into consideration when evaluating the cost effectiveness of such a scheme.

2.2.4 An Alternative Partitioning Algorithm

Instead of applying SLF-transformations as shown in (2.9)-(2.12), cyclic reduction can be used in a sequential fashion on each partition. To illustrate, let $m_0 = 7$ in (2.9), so the first partition appears as

$$\left[\begin{array}{ccccccccc|c}
S_0 & T_0 & & & & & & & & \phi_0 \\
 & S_1 & T_1 & & & & & & & \phi_1 \\
 & & S_2 & T_2 & & & & & & \phi_2 \\
 & & & S_3 & T_3 & & & & & \phi_3 \\
 & & & & S_4 & T_4 & & & & \phi_4 \\
 & & & & & S_5 & T_5 & & & \phi_5 \\
 & & & & & & S_6 & T_6 & & \phi_6 \\
 & & & & & & & S_7 & T_7 & \phi_7
\end{array}\right] \tag{2.16}$$

Since only one processor is available for this partition, the reduction requires 7 steps as opposed to 3 sweeps in Figure 2.1. After 4 steps (2.16) is transformed to

$$\left[\begin{array}{ccccccccc|c}
S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & & & & \phi_0^{(1)} \\
S_1^{(1)} & & T_1^{(1)} & & & & & & & \phi_1^{(1)} \\
 & & S_2^{(2)} & R_2^{(2)} & T_2^{(2)} & & & & & \phi_2^{(2)} \\
 & & S_3^{(2)} & & T_3^{(2)} & & & & & \phi_3^{(2)} \\
 & & & & S_4^{(3)} & R_4^{(3)} & T_4^{(3)} & & & \phi_4^{(3)} \\
 & & & & S_5^{(3)} & & T_5^{(3)} & & & \phi_5^{(3)} \\
 & & & & & & S_6^{(4)} & R_6^{(4)} & T_6^{(4)} & \phi_6^{(4)} \\
 & & & & & & S_7^{(4)} & & T_7^{(4)} & \phi_7^{(4)}
\end{array}\right] \tag{2.17}$$

After 2 more steps (2.17) is transformed to

$$\left[\begin{array}{ccccccccc|c}
S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & & & & \phi_0^{(1)} \\
\boxed{S_1^{(5)}} & & \boxed{R_1^{(5)}} & & \boxed{T_1^{(5)}} & & & & & \boxed{\phi_1^{(5)}} \\
 & & S_2^{(2)} & R_2^{(2)} & T_2^{(2)} & & & & & \phi_2^{(2)} \\
\boxed{S_3^{(5)}} & & & & \boxed{T_3^{(5)}} & & & & & \boxed{\phi_3^{(5)}} \\
 & & & & S_4^{(3)} & R_4^{(3)} & T_4^{(3)} & & & \phi_4^{(3)} \\
 & & & & \boxed{S_5^{(6)}} & & \boxed{R_5^{(6)}} & & \boxed{T_5^{(6)}} & \boxed{\phi_5^{(6)}} \\
 & & & & & & S_6^{(4)} & R_6^{(4)} & T_6^{(4)} & \phi_6^{(4)} \\
 & & & & \boxed{S_7^{(6)}} & & & & \boxed{T_7^{(6)}} & \boxed{\phi_7^{(6)}}
\end{array}\right] \tag{2.18}$$

After the final step the partition has the form

$$\left[\begin{array}{ccccccccc|c}
S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & & & & \phi_0^{(1)} \\
S_1^{(5)} & & R_1^{(5)} & & T_1^{(5)} & & & & & \phi_1^{(5)} \\
 & & S_2^{(2)} & R_2^{(2)} & T_2^{(2)} & & & & & \phi_2^{(2)} \\
\boxed{S_3^{(7)}} & & & & \boxed{R_3^{(7)}} & & & & \boxed{T_3^{(7)}} & \boxed{\phi_3^{(7)}} \\
 & & & & S_4^{(3)} & R_4^{(3)} & T_4^{(3)} & & & \phi_4^{(3)} \\
 & & & & S_5^{(6)} & & R_5^{(6)} & & T_5^{(6)} & \phi_5^{(6)} \\
 & & & & & & S_6^{(4)} & R_6^{(4)} & T_6^{(4)} & \phi_6^{(4)} \\
\boxed{S_7^{(7)}} & & & & & & & & \boxed{T_7^{(7)}} & \boxed{\phi_7^{(7)}}
\end{array}\right] \tag{2.19}$$

The end mesh variables of this partition, y_0 and y_8, are computed by solving a compacted system constructed from the last block-row of each partition as before (Figures 2.1 and 2.2). Once the end variables are known, the interior variables are computed in 7 back-substitution steps similar to (2.13), except that here the sequence is block-row 3,1,5,0,2,4,6.

The cost of applying cyclic reduction to each partition is the same as that of the algorithm in §2.2.3. Which algorithm is better in terms of stability, however, is currently not known. As first pointed out in [Wrig 93] and as shown by several additional numerical experiments in Chapter 3 of this thesis, *SLF*-LU when implemented as described in §2.2.3 is potentially unstable on certain problems when the partitions become large. Whether or not cyclic reduction is a more stable alternative for these problems is an open question.

2.3 Global Stability Control

As shown in the previous section, SLF-transformations can be used with partitioning and cyclic reduction to provide a stable algorithm for solving ABD systems. With as few as $M/\log M$ processors, a parallel complexity of $\mathcal{O}(\log M)$ block-steps is attainable. Theoretically this is the optimal speed-up with respect to M. Nevertheless, this algorithm can still be improved by reducing the local operation counts (with respect to n) and storage requirements. In this section we propose a new algorithm, RSCALE, based on a new technique: *eigenvalue rescaling*. As will be shown in §4.1, RSCALE requires fewer local operations than either of the SLF-based algorithms. When used to solve ABD systems of the form (1.9), RSCALE can be up to twice as fast as *SLF*-LU—the fastest of the two SLF-based algorithms. When used to solve ABD systems of the form (1.8), RSCALE is marginally faster than *SLF*-LU on most problems, and often has better stability properties (Chapter 3).

2.3.1 Improving SLF Transformations

Three ways in which SLF-transformations differ from the elementary transformation used in block cyclic reduction (2.6) are:

1. the $n \times n$ block $T_i^{(k)}$ in (2.7) and (2.8) fills-in requiring extra storage and resulting in a more costly back-solve,

2. a $2n \times n$ matrix factorization followed by four $n \times n$ matrix products is required, as opposed to a single $n \times n$ matrix product, and

3. the identity matrix that appears in the third form of the Jacobian (1.9) is destroyed, resulting in additional fill-in if this form arises naturally from the discretization.

Indeed, the first two differences also hold when comparing SLF-transformations to the block-step used in state-of-the-art sequential codes such as COLROW, except that the local operation cost in COLROW is not due to matrix multiplication.

So the advantages of block cyclic reduction are obvious—it is faster and requires less storage than SLF-transformations. The drawback, of course, is that it is not necessarily stable when used to solve ABD systems arising from numerical methods for BVPs. The algorithms discussed in §2.1 *are* stable, though, when used to solve certain classes of linear recurrence relations. In particular, consider the two-term recurrence

$$y_0 = \alpha, \quad y_{i+1} = V_i y_i + \phi_i \quad i = 0, 1, \ldots \tag{2.20}$$

where $\|V_i\|_2 \leq 1$ for all i. This recurrence is stable and can be computed in parallel using cyclic reduction ([Sche 84]). In fact, this is precisely the recurrence that arises when solving a stable linear IVP. As noted in [Gear 88], fast methods for the solution of linear IVPs can be constructed from fast algorithms for solving (2.20).

Now, if the ABD system (1.9) could be transformed and the equations recast as (2.20), the analysis in [Sche 84] and [Gear 88] would apply and block cyclic reduction as described in §2.1 could be used on this problem. In order to be competitive with SLF the transformation cannot be expensive, and it should not have an adverse effect on the condition of the system (c.f. the normal equations method in [Asch 91]). The RSCALE algorithm described below is based on such a transformation. Since the mesh variables are changed globally to control stability during the reduction, we refer to this technique as *global stability control*.

2.3.2 Sequential Rescaling

This form of the algorithm uses a sequential mesh variable transformation to produce an ABD system that can be solved stably using block cyclic reduction. Instead of solving $\mathcal{J}Y = \Phi$ for Y, $(\mathcal{J}C)\tilde{Y} = \Phi$ is solved for $\tilde{Y} = C^{-1}Y$, where

$$C = \begin{bmatrix} I & -\sigma I & & & & \\ & I & -\sigma I & & & \\ & & I & -\sigma I & & \\ & & & \ddots & \ddots & \\ & & & & I & -\sigma I \\ & & & & & I \end{bmatrix}, \quad C^{-1} = \begin{bmatrix} I & \sigma I & \sigma^2 I & \sigma^3 I & \cdots & \sigma^M I \\ & I & \sigma I & \sigma^2 I & \cdots & \sigma^{M-1} I \\ & & I & \sigma I & \cdots & \sigma^{M-2} I \\ & & & \ddots & \ddots & \vdots \\ & & & & I & \sigma I \\ & & & & & I \end{bmatrix} \tag{2.21}$$

For ease of notation, $\sigma = 1$ is assumed throughout most of this section. The optimal choice of σ is currently an open question. Numerical experiments show that $\sigma = 1$ is appropriate for most test problems. Varying σ, though, can affect the accuracy of the solution, as is discussed further in [Panc 92] and Chapter 3.

Once $\tilde{Y}$ has been computed, the original mesh variables can be recovered easily:

$$y_i = \tilde{y}_i - \tilde{y}_{i+1}, \quad i = 0, \ldots, M-2. \tag{2.22}$$

The algorithm proceeds as follows—first, the system is transformed (the right-hand side is not affected by the mesh variable transformation):

$$\left[\begin{array}{c} \begin{bmatrix} \mathcal{B}_a & & & & & \mathcal{B}_b \\ V_0 & I & & & & \\ & V_1 & I & & & \\ & & \ddots & \ddots & & \\ & & & V_{M-3} & I & \\ & & & & V_{M-2} & I \\ & & & & & V_{M-1} \quad I \end{bmatrix} \times \begin{bmatrix} I & -I & & & & \\ & I & -I & & & \\ & & I & -I & & \\ & & & \ddots & \ddots & \\ & & & & I & -I \\ & & & & & I \end{bmatrix} \end{array} \middle| \begin{array}{c} g \\ \phi_0 \\ \phi_1 \\ \vdots \\ \phi_{M-3} \\ \phi_{M-2} \\ \phi_{M-1} \end{array} \right]$$

$$= \left[\begin{array}{cccccccc|c}
\mathcal{B}_a & -\mathcal{B}_a & & & & & & \mathcal{B}_b & g \\
V_0 & I\text{–}V_0 & -I & & & & & & \phi_0 \\
 & V_1 & I\text{–}V_1 & -I & & & & & \phi_1 \\
 & & \ddots & \ddots & \ddots & & & & \vdots \\
 & & & V_{M-3} & I\text{–}V_{M-3} & -I & & & \phi_{M-3} \\
 & & & & V_{M-2} & I\text{–}V_{M-2} & -I & & \phi_{M-2} \\
 & & & & & V_{M-1} & I\text{–}V_{M-1} & & \phi_{M-1}
\end{array}\right] \tag{2.23}$$

This requires no extra storage and only minimal computation in the actual implementation due to the simple form of C. Now (2.23) is no longer ABD, but it can be transformed back to that form by a sequential process starting at the bottom block-row and working up. First, block-row $M-1$ is multiplied through by $(I - V_{M-1})^{-1}$ giving

$$\left[\begin{array}{cccccccc|c}
\mathcal{B}_a & -\mathcal{B}_a & & & & & & \mathcal{B}_b & g \\
V_0 & I\text{–}V_0 & -I & & & & & & \phi_0 \\
 & V_1 & I\text{–}V_1 & -I & & & & & \phi_1 \\
 & & \ddots & \ddots & \ddots & & & & \vdots \\
 & & & V_{M-3} & I\text{–}V_{M-3} & -I & & & \phi_{M-3} \\
 & & & & V_{M-2} & I\text{–}V_{M-2} & -I & & \phi_{M-2} \\
 & & & & & \boxed{V_{M-1}^{(1)}} & \boxed{I} & & \boxed{\phi_{M-1}^{(1)}}
\end{array}\right] \tag{2.24}$$

Next, block-row $M-1$ is added to block-row $M-2$ to annihilate the $-I$ in block-row $M-2$. Then block-row $M-2$ is multiplied through by $(I - V_{M-2} + V_{M-1}^{(1)})^{-1}$ giving

$$\left[\begin{array}{cccccccc|c}
\mathcal{B}_a & -\mathcal{B}_a & & & & & & \mathcal{B}_b & g \\
V_0 & I\text{–}V_0 & -I & & & & & & \phi_0 \\
 & V_1 & I\text{–}V_1 & -I & & & & & \phi_1 \\
 & & \ddots & \ddots & \ddots & & & & \vdots \\
 & & & V_{M-3} & I\text{–}V_{M-3} & -I & & & \phi_{M-3} \\
 & & & & \boxed{V_{M-2}^{(2)}} & \boxed{I} & & & \boxed{\phi_{M-2}^{(2)}} \\
 & & & & & V_{M-1}^{(1)} & I & & \phi_{M-1}^{(1)}
\end{array}\right] \tag{2.25}$$

Block-rows $M-3,\ldots,0$ are transformed similarly giving

$$\left[\begin{array}{ccccccc|c}
\mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b & g \\
\boxed{V_0^{(M)}} & \boxed{I} & & & & & & \boxed{\phi_0^{(M)}} \\
 & V_1^{(M-1)} & I & & & & & \phi_1^{(M-1)} \\
 & & \ddots & \ddots & & & & \vdots \\
 & & & V_{M-3}^{(3)} & I & & & \phi_{M-3}^{(3)} \\
 & & & & V_{M-2}^{(2)} & I & & \phi_{M-2}^{(2)} \\
 & & & & & V_{M-1}^{(1)} & I & \phi_{M-1}^{(1)}
\end{array}\right] \tag{2.26}$$

Finally, $-\mathcal{B}_a$ in the top block-row is annihilated:

$$\left[\begin{array}{ccccccc|c}
\boxed{\mathcal{B}_a^{(M+1)}} & & & & & & \mathcal{B}_b & \boxed{g^{(M+1)}} \\
V_0^{(M)} & I & & & & & & \phi_0^{(M)} \\
 & V_1^{(M-1)} & I & & & & & \phi_1^{(M-1)} \\
 & & \ddots & \ddots & & & & \vdots \\
 & & & V_{M-3}^{(3)} & I & & & \phi_{M-3}^{(3)} \\
 & & & & V_{M-2}^{(2)} & I & & \phi_{M-2}^{(2)} \\
 & & & & & V_{M-1}^{(1)} & I & \phi_{M-1}^{(1)}
\end{array}\right] \tag{2.27}$$

This last system is once again ABD. More importantly, numerical experiments in [Panc 92] and Chapter 4 and a preliminary analysis in Chapter 3 indicate that it can be solved stably using the algorithms in §2.1 because, for well-conditioned problems, the above transformation effectively rescales the eigenvalues λ and the norm of each V_i so that

1. $|\lambda| \in (0, 1+\epsilon_1)$, and
2. $\|V_i\|_2 \in (1-\epsilon_2, 1+\epsilon_3)$

where $\epsilon_1,\ \epsilon_2$, and ϵ_3 can be made arbitrarily small with a suitable choice for σ in (2.21). For the test problems considered in [Panc 92] and Chapter 4, the choice of $\sigma = 1$ has been sufficient for stability. In addition, even though theoretically the transformation may fail for any given σ if $(I - \sigma(V_i + V_{i+1}^{(k)}))$ becomes numerically singular for some i, numerical experiments have shown that a dynamic shift in σ at this stage can redistribute the eigenvalues and allow the algorithm to continue. At the time of writing this thesis, the only ABD matrices known to require this dynamic shift are themselves very poorly conditioned. These issues are discussed further in Chapter 3.

The total operation count for this algorithm—including operations required for the mesh variable transformation, cyclic reduction, back-solve and mesh variable recovery (2.22)—is

approximately half that of *SLF*-LU. Multiple right-hand sides can be handled in $\mathcal{O}(Mn^2)$ time if the local factorizations of $(I - \sigma(V_i + V_{i+1}^{(k)}))$ are stored. In fact, once the Jacobian has been reduced, the time required to solve with a different right-hand side is comparable to that of algorithms based on alternate row and column pivoting. See Chapter 4 for details.

RSCALE requires $2Mn^2$ storage if the local factorizations are kept—the same as COLROW and half that of SLF-based algorithms. Since RSCALE is indifferent to the separability of the boundary conditions, systems with coupled BCs can be solved without fill-in.

2.3.3 Parallel Rescaling

Although (2.27) can be solved in parallel, constructing it as shown in (2.24)-(2.27) is a sequential process. This section explains how the mesh variable transformation can be integrated with partitioning and cyclic reduction to provide a parallel RSCALE algorithm.

As in §2.2.3, the ABD system is first partitioned into P blocks of M/P block-rows each. The transformation C is modified slightly by omitting the $-I$ at the boundary between each partition. For example, the transformation across the first partition has the form

$$\left[\begin{array}{c|c|c} \begin{bmatrix} \ddots & \ddots & & & & \\ V_{m_0-2} & I & & & & \\ & V_{m_0-1} & I & & & \\ & & V_{m_0} & I & & \\ & & & V_{m_0+1} & I & \\ & & & & V_{m_0+2} & I \\ & & & & & \ddots \; \ddots \end{bmatrix} & \times & \begin{bmatrix} \ddots & \ddots & & & & \\ I & -I & & & & \\ & I & -I & & & \\ & & I & & & \\ & & & I & -I & \\ & & & & I & -I \\ & & & & & \ddots \; \ddots \end{bmatrix} & \begin{matrix} \vdots \\ \phi_{m_0-2} \\ \phi_{m_0-1} \\ \phi_{m_0} \\ \phi_{m_0+1} \\ \phi_{m_0+2} \\ \vdots \end{matrix} \end{array}\right]$$

$$= \left[\begin{array}{cccccccc|c} \ddots & \ddots & \ddots & & & & & & \vdots \\ V_{m_0-2} & I\text{–}V_{m_0-2} & -I & & & & & & \phi_{m_0-2} \\ & V_{m_0-1} & I\text{–}V_{m_0-1} & & & & & & \phi_{m_0-1} \\ & & V_{m_0} & I & -I & & & & \phi_{m_0} \\ & & & V_{m_0+1} & I\text{–}V_{m_0+1} & -I & & & \phi_{m_0+1} \\ & & & & V_{m_0+2} & I\text{–}V_{m_0+2} & -I & & \phi_{m_0+2} \\ & & & & & \ddots & \ddots & \ddots & \vdots \end{array}\right] \tag{2.28}$$

The steps shown below on the first partition can be carried out simultaneously on all P partitions. Processing starts at block-row $m_0 - 1$ and works up. The partition is changed back to

ABD form as in the sequential algorithm, but now V_{m_0} is "dragged" to the left as well; i.e. the reduction phase is interleaved with the rescaling. (This also could be done in the sequential algorithm.) So the first step consists of

1. $V^{(1)}_{m_0-1} = (I - V_{m_0-1})^{-1} V_{m_0-1}, \quad \phi^{(1)}_{m_0-1} = (I - V_{m_0-1})^{-1} \phi_{m_0-1}$,
2. $V^{(1)}_{m_0} = -V_{m_0} V^{(1)}_{m_0-1}, \quad \phi^{(1)}_{m_0} = \phi_{m_0} - V_{m_0} \phi^{(1)}_{m_0-1}$.

After the first step the partition is

$$\left[\begin{array}{ccccccccc|c}
\ddots & \ddots & \ddots & & & & & & & \vdots \\
V_{m_0-2} & I\text{–}V_{m_0-2} & -I & & & & & & & \phi_{m_0-2} \\
 & \boxed{V^{(1)}_{m_0-1}} & \boxed{I} & & & & & & & \boxed{\phi^{(1)}_{m_0-1}} \\
 & \boxed{V^{(1)}_{m_0}} & & I & -I & & & & & \boxed{\phi^{(1)}_{m_0}} \\
 & & & V_{m_0+1} & I\text{–}V_{m_0+1} & -I & & & & \phi_{m_0+1} \\
 & & & & V_{m_0+2} & I\text{–}V_{m_0+2} & -I & & & \phi_{m_0+2} \\
 & & & & & \ddots & \ddots & \ddots & & \vdots
\end{array}\right] \tag{2.29}$$

Next, the $-I$ in block-row $m_0 - 2$ is annihilated, block-row $m_0 - 2$ is multiplied through by $(I - V_{m_0-2} + V^{(1)}_{m_0-1})^{-1}$, and $V^{(1)}_{m_0}$ is dragged one block-column to the left:

$$\left[\begin{array}{ccccccccc|c}
\ddots & \ddots & \ddots & & & & & & & \vdots \\
 & \boxed{V^{(2)}_{m_0-2}} & \boxed{I} & & & & & & & \boxed{\phi^{(2)}_{m_0-2}} \\
 & & V^{(1)}_{m_0-1} & I & & & & & & \phi^{(1)}_{m_0-1} \\
 & \boxed{V^{(2)}_{m_0}} & & & I & -I & & & & \boxed{\phi^{(2)}_{m_0}} \\
 & & & & V_{m_0+1} & I\text{–}V_{m_0+1} & -I & & & \phi_{m_0+1} \\
 & & & & & V_{m_0+2} & I\text{–}V_{m_0+2} & -I & & \phi_{m_0+2} \\
 & & & & & & \ddots & \ddots & \ddots & \vdots
\end{array}\right] \tag{2.30}$$

Block-rows $m_0 - 3, \ldots, 0$ are transformed similarly. Assuming $m_0 = m_1$, the second partition will have been transformed by this time as well:

$$\left[\begin{array}{ccccccccc|c}
\mathcal{B}_a & -\mathcal{B}_a & & & & & & & & g \\
\boxed{V^{(m_0)}_0} & \boxed{I} & & & & & & & & \boxed{\phi^{(m_0)}_0} \\
 & \ddots & \ddots & & & & & & & \vdots \\
 & & V^{(1)}_{m_0-1} & I & & & & & & \phi^{(1)}_{m_0-1} \\
\boxed{V^{(m_0)}_{m_0}} & & & & I & -I & & & & \boxed{\phi^{(m_0)}_{m_0}} \\
 & & & & \boxed{V^{(m_1)}_{m_0+1}} & \boxed{I} & & & & \boxed{\phi^{(m_1)}_{m_0+1}} \\
 & & & & & V^{(m_1-1)}_{m_0+2} & I & & & \phi^{(m_1-1)}_{m_0+2} \\
 & & & & & & \ddots & \ddots & & \vdots
\end{array}\right] \tag{2.31}$$

Finally, $-\mathcal{B}_a$ in the top block-row is annihilated, as is the $-I$ appearing in the bottom block-row of each partition (except the last):

$$\left[\begin{array}{ccccccc|c}
\boxed{\mathcal{B}_a^{(m_0+1)}} & & & & & & & \boxed{g^{(m_0+1)}} \\
V_0^{(m_0)} & I & & & & & & \phi_0^{(m_0)} \\
& \ddots & \ddots & & & & & \vdots \\
& & V_{m_0-1}^{(1)} & I & & & & \phi_{m_0-1}^{(1)} \\
V_{m_0}^{(m_0)} & & & \boxed{W_{m_0}^{(m_1+1)}} & & & & \boxed{\phi_{m_0}^{(m_1+1)}} \\
& & & V_{m_0+1}^{(m_1)} & I & & & \phi_{m_0+1}^{(m_1)} \\
& & & & V_{m_0+2}^{(m_1-1)} & I & & \phi_{m_0+2}^{(m_1-1)} \\
& & & & & \ddots & \ddots & \vdots
\end{array}\right] \tag{2.32}$$

where $W_{m_0}^{(m_1+1)} = I + V_{m_0+1}^{(m_1)}$, $\phi_{m_0}^{(m_1+1)} = \phi_{m_0}^{(m_0)} + \phi_{m_0+1}^{(m_1)}$.

$\tilde{y}_0$ and $\tilde{y}_{m_0+1}$ are the end mesh variables of the first partition. Once they are known, the interior variables can be computed by a forward recurrence and the original mesh variables can be restored:

$$\tilde{y}_{i+1} = \phi_i^{(k)} - V_i^{(k)}\tilde{y}_i, \quad y_i = \tilde{y}_i - \tilde{y}_{i+1}, \quad i = 0, \ldots, m_0 - 1 \tag{2.33}$$

$$y_{m_0} = \tilde{y}_{m_0} - \tilde{y}_{m_0+1}, \quad y_{m_0+1} = \tilde{y}_{m_0+1}$$

As with (2.27), numerical experiments show that the eigenvalues and norm of each $V_i^{(k)}$ are rescaled by the above transformation allowing (2.33) to be computed stably.

The end mesh variables for each partition are computed by solving a compacted system of order P constructed from the last block-row of each partition. Letting

$$[\mathcal{B}_a^{(m_0+1)} g^{(m_0+1)}] \equiv [\hat{\mathcal{B}}_a \ \hat{g}], \quad [V_{m_i}^{(m_i)} W_{m_i}^{(m_i+1)} \phi_{m_i}^{(m_i+1)}] \equiv [V_{p_i} \ W_{p_i} \ \phi_{p_i}], \quad i = 0 \ldots P-1,$$

this compacted system has the form

$$\left[\begin{array}{ccccccc|c}
\hat{\mathcal{B}}_a & & & & & & \mathcal{B}_b & \hat{g} \\
V_{p_0} & W_{p_0} & & & & & & \phi_{p_0} \\
& V_{p_1} & W_{p_1} & & & & & \phi_{p_1} \\
& & \ddots & \ddots & & & & \vdots \\
& & & V_{p_{P-2}} & W_{p_{P-2}} & & & \phi_{p_{P-2}} \\
& & & & V_{p_{P-1}} & I & & \phi_{p_{P-1}}
\end{array}\right] \tag{2.34}$$

This system has a slightly different structure than (1.9)—the identity is replaced by W_{p_i} in rows $0 \ldots P-2$. Unlike (1.8), (2.34) *cannot* be transformed by multiplying each block-row through

by $W_{p_i}^{-1}$, because W_{p_i} is not necessarily well-conditioned.[3] Nevertheless, mesh variable transformation can still be used in combination with cyclic reduction to solve this system stably in $\mathcal{O}(\log P)$ block-steps. Two versions of the algorithm are described below (again, $P = 8$ is used to simplify the figures).

The first version requires $\log P$ reduction sweeps and $\log P$ back-solve sweeps. During the first reduction sweep, (2.34) is transformed as follows:

$$\left[\begin{array}{c|c} \begin{bmatrix} \hat{\mathcal{B}}_a & & & & & & & \mathcal{B}_b \\ V_{p_0} & W_{p_0} & & & & & & \\ & V_{p_1} & W_{p_1} & & & & & \\ & & V_{p_2} & W_{p_2} & & & & \\ & & & V_{p_3} & W_{p_3} & & & \\ & & & & V_{p_4} & W_{p_4} & & \\ & & & & & V_{p_5} & W_{p_5} & \\ & & & & & & V_{p_6} & W_{p_6} \\ & & & & & & & V_{p_7} \quad I \end{bmatrix} & \begin{bmatrix} \hat{g} \\ \phi_{p_0} \\ \phi_{p_1} \\ \phi_{p_2} \\ \phi_{p_3} \\ \phi_{p_4} \\ \phi_{p_5} \\ \phi_{p_6} \\ \phi_{p_7} \end{bmatrix} \end{array}\right] \times \left[\begin{array}{c|c} \begin{bmatrix} I & -I & & & & & & \\ & I & & & & & & \\ & & I & -I & & & & \\ & & & I & & & & \\ & & & & I & -I & & \\ & & & & & I & & \\ & & & & & & I & -I \\ & & & & & & & I \end{bmatrix} & \begin{bmatrix} \hat{g} \\ \phi_{p_0} \\ \phi_{p_1} \\ \phi_{p_2} \\ \phi_{p_3} \\ \phi_{p_4} \\ \phi_{p_5} \\ \phi_{p_6} \\ \phi_{p_7} \end{bmatrix} \end{array}\right]$$

$$= \left[\begin{array}{c|c} \begin{bmatrix} \hat{\mathcal{B}}_a & -\hat{\mathcal{B}}_a & & & & & & & \mathcal{B}_b \\ V_{p_0} & W_{p_0}{-}V_{p_0} & & & & & & & \\ & V_{p_1} & W_{p_1} & -W_{p_1} & & & & & \\ & & V_{p_2} & W_{p_2}{-}V_{p_2} & & & & & \\ & & & V_{p_3} & W_{p_3} & -W_{p_3} & & & \\ & & & & V_{p_4} & W_{p_4}{-}V_{p_4} & & & \\ & & & & & V_{p_5} & W_{p_5} & -W_{p_5} & \\ & & & & & & V_{p_6} & W_{p_6}{-}V_{p_6} & \\ & & & & & & & V_{p_7} \quad I \end{bmatrix} & \begin{bmatrix} \hat{g} \\ \phi_{p_0} \\ \phi_{p_1} \\ \phi_{p_2} \\ \phi_{p_3} \\ \phi_{p_4} \\ \phi_{p_5} \\ \phi_{p_6} \\ \phi_{p_7} \end{bmatrix} \end{array}\right] \tag{2.35}$$

This transformation *reconditions* $(W_{p_i} - V_{p_i}), i = 0, 2, 4, 6$, so these block-rows can now be multiplied through by $(W_{p_i} - V_{p_i})^{-1}$. For $i = 0, 2, 4, 6$, each of 4 processors is active during the first sweep computing

1. $V_{p_i}^{(1)} = (W_{p_i} - V_{p_i})^{-1} V_{p_i}, \quad \phi_{p_i}^{(1)} = (W_{p_i} - V_{p_i})^{-1} \phi_{p_i}$,

2. $V_{p_{i+1}}^{(1)} = -V_{p_{i+1}} V_{p_i}^{(1)}, \quad \phi_{p_{i+1}}^{(1)} = \phi_{p_{i+1}} - V_{p_{i+1}} \phi_{p_i}^{(1)}$.

[3] The same is true for T_{p_i} in (2.14).

In addition, $-\hat{\mathcal{B}}_a$ and $-W_{p_i}, i = 1, 3, 5$, are annihilated, resulting in a system of the form

$$\left[\begin{array}{ccccccccc|c}
\hat{\mathcal{B}}_a^{(1)} & & & & & & & & \mathcal{B}_b & \hat{g}^{(1)} \\
V_{p_0}^{(1)} & I & & & & & & & & \phi_{p_0}^{(1)} \\
V_{p_1}^{(1)} & & W_{p_1}^{(1)} & & & & & & & \phi_{p_1}^{(1)} \\
 & & V_{p_2}^{(1)} & I & & & & & & \phi_{p_2}^{(1)} \\
 & & V_{p_3}^{(1)} & & W_{p_3}^{(1)} & & & & & \phi_{p_3}^{(1)} \\
 & & & & V_{p_4}^{(1)} & I & & & & \phi_{p_4}^{(1)} \\
 & & & & V_{p_5}^{(1)} & & W_{p_5}^{(1)} & & & \phi_{p_5}^{(1)} \\
 & & & & & & V_{p_6}^{(1)} & I & & \phi_{p_6}^{(1)} \\
 & & & & & & V_{p_7}^{(1)} & & I & \phi_{p_7}^{(1)}
\end{array}\right] \tag{2.36}$$

This uncouples the odd-indexed variables from the system. Once $\tilde{y}_{p_i}, i = 0, 2, 4, 6, 8$ are known, $\tilde{y}_{p_{i+1}}, i = 0, 2, 4, 6$ can be computed and the transformation in (2.35) reversed by

$$\tilde{y}_{p_{i+1}} = \phi_{p_i}^{(1)} - V_{p_i}^{(1)} \tilde{y}_{p_i}, \quad y_{p_i} = \tilde{y}_{p_i} - \tilde{y}_{p_{i+1}}, \quad y_{p_{i+1}} = \tilde{y}_{p_{i+1}}, \quad i = 0, 2, 4, 6. \tag{2.37}$$

The even-indexed variables are computed during the second sweep by solving a reduced system constructed from the odd rows and columns of (2.36). This system together with the transformation that is applied during the second sweep is shown below:

$$\left[\begin{array}{c}
\left[\begin{array}{ccccc|c}
\hat{\mathcal{B}}_a^{(1)} & & & & \mathcal{B}_b & \hat{g}^{(1)} \\
V_{p_1}^{(1)} & W_{p_1}^{(1)} & & & & \phi_{p_1}^{(1)} \\
 & V_{p_3}^{(1)} & W_{p_3}^{(1)} & & & \phi_{p_3}^{(1)} \\
 & & V_{p_5}^{(1)} & W_{p_5}^{(1)} & & \phi_{p_5}^{(1)} \\
 & & & V_{p_7}^{(1)} & I & \phi_{p_7}^{(1)}
\end{array}\right] \times \left[\begin{array}{ccccc}
I & -I & & & \\
 & I & & & \\
 & & I & -I & \\
 & & & I & \\
 & & & & I
\end{array}\right]
\end{array}\right]$$

$$= \left[\begin{array}{ccccc|c}
\hat{\mathcal{B}}_a^{(1)} & -\hat{\mathcal{B}}_a^{(1)} & & & \mathcal{B}_b & \hat{g}^{(1)} \\
V_{p_1}^{(1)} & W_{p_1}^{(1)} - V_{p_1}^{(1)} & & & & \phi_{p_1}^{(1)} \\
 & V_{p_3}^{(1)} & W_{p_3}^{(1)} & -W_{p_3}^{(1)} & & \phi_{p_3}^{(1)} \\
 & & V_{p_5}^{(1)} & W_{p_5}^{(1)} - V_{p_5}^{(1)} & & \phi_{p_5}^{(1)} \\
 & & & V_{p_7}^{(1)} & I & \phi_{p_7}^{(1)}
\end{array}\right] \tag{2.38}$$

The algorithm continues recursively for $\log P$ sweeps resulting in a final $2n \times 2n$ compacted system. This system is solved sequentially by a stable method, such as Gaussian elimination, followed by $\log P$ back-solve/transformation reversal sweeps of the form (2.37).

Figure 2.3: Cyclic reduction to Λ-shape Jacobian using rescaling

Sweep #1

$$\begin{bmatrix}
\hat{\mathcal{B}}_a & -\hat{\mathcal{B}}_a & & & & & & & & \mathcal{B}_b \\
V_{p_0} & W_{p_0}{-}V_{p_0} & & & & & & & & \\
 & V_{p_1} & W_{p_1} & -W_{p_1} & & & & & & \\
 & & V_{p_2} & W_{p_2}{-}V_{p_2} & & & & & & \\
 & & & V_{p_3} & W_{p_3} & -W_{p_3} & & & & \\
 & & & & V_{p_4} & W_{p_4}{-}V_{p_4} & & & & \\
 & & & & & V_{p_5} & W_{p_5} & -W_{p_5} & & \\
 & & & & & & V_{p_6} & W_{p_6}{-}V_{p_6} & & \\
 & & & & & & & V_{p_7} & I &
\end{bmatrix} \longrightarrow \begin{bmatrix}
\hat{\mathcal{B}}_a^{(1)} & & & & & & & & & \mathcal{B}_b \\
V_{p_0}^{(1)} & I & & & & & & & & \\
V_{p_1}^{(1)} & & W_{p_1}^{(1)} & & & & & & & \\
 & & V_{p_2}^{(1)} & I & & & & & & \\
 & & V_{p_3}^{(1)} & & W_{p_3}^{(1)} & & & & & \\
 & & & & V_{p_4}^{(1)} & I & & & & \\
 & & & & V_{p_5}^{(1)} & & W_{p_5}^{(1)} & & & \\
 & & & & & & V_{p_6}^{(1)} & I & & \\
 & & & & & & V_{p_7}^{(1)} & & I &
\end{bmatrix}$$

Sweep #2

$$\begin{bmatrix}
\hat{\mathcal{B}}_a^{(1)} & & -\hat{\mathcal{B}}_a^{(1)} & & & & & & & \mathcal{B}_b \\
V_{p_0}^{(1)} & I & -V_{p_0}^{(1)} & & & & & & & \\
V_{p_1}^{(1)} & & W_{p_1}^{(1)}{-}V_{p_1}^{(1)} & & & & & & & \\
 & & V_{p_2}^{(1)} & I & & & & & & \\
 & & V_{p_3}^{(1)} & & W_{p_3}^{(1)} & -W_{p_3}^{(1)} & & & & \\
 & & & & V_{p_4}^{(1)} & I \; -V_{p_4}^{(1)} & & & & \\
 & & & & V_{p_5}^{(1)} & & W_{p_5}^{(1)}{-}V_{p_5}^{(1)} & & & \\
 & & & & & & V_{p_6}^{(1)} & I & & \\
 & & & & & & V_{p_7}^{(1)} & & I &
\end{bmatrix} \longrightarrow \begin{bmatrix}
\hat{\mathcal{B}}_a^{(2)} & & & & & & & & & \mathcal{B}_b \\
V_{p_0}^{(2)} & I & & & & & & & & \\
V_{p_1}^{(2)} & & I & & & & & & & \\
V_{p_2}^{(2)} & & & I & & & & & & \\
V_{p_3}^{(2)} & & & & W_{p_3}^{(2)} & & & & & \\
 & & & & V_{p_4}^{(2)} & I & & & & \\
 & & & & V_{p_5}^{(2)} & & I & & & \\
 & & & & V_{p_6}^{(2)} & & & I & & \\
 & & & & V_{p_7}^{(2)} & & & & I &
\end{bmatrix}$$

Sweep #3

$$\begin{bmatrix}
\hat{\mathcal{B}}_a^{(2)} & & & -\hat{\mathcal{B}}_a^{(2)} & & & & & & \mathcal{B}_b \\
V_{p_0}^{(2)} & I & & -V_{p_0}^{(2)} & & & & & & \\
V_{p_1}^{(2)} & & I & -V_{p_1}^{(2)} & & & & & & \\
V_{p_2}^{(2)} & & & I \; -V_{p_2}^{(2)} & & & & & & \\
V_{p_3}^{(2)} & & & & W_{p_3}^{(2)}{-}V_{p_3}^{(2)} & & & & & \\
 & & & & V_{p_4}^{(2)} & I & & & & \\
 & & & & V_{p_5}^{(2)} & & I & & & \\
 & & & & V_{p_6}^{(2)} & & & I & & \\
 & & & & V_{p_7}^{(2)} & & & & I &
\end{bmatrix} \longrightarrow \begin{bmatrix}
\hat{\mathcal{B}}_a^{(3)} & & & & & & & & & \mathcal{B}_b \\
V_{p_0}^{(3)} & I & & & & & & & & \\
V_{p_1}^{(3)} & & I & & & & & & & \\
V_{p_2}^{(3)} & & & I & & & & & & \\
V_{p_3}^{(3)} & & & & I & & & & & \\
V_{p_4}^{(3)} & & & & & I & & & & \\
V_{p_5}^{(3)} & & & & & & I & & & \\
V_{p_6}^{(3)} & & & & & & & I & & \\
V_{p_7}^{(3)} & & & & & & & & I &
\end{bmatrix}$$

Figure 2.4: Mesh variable transformation used prior to each sweep in Figure 2.3

Sweep #1

$$C_1 = \begin{bmatrix} I & -I & & & & & & \\ & I & & & & & & \\ & & I & -I & & & & \\ & & & I & & & & \\ & & & & I & -I & & \\ & & & & & I & & \\ & & & & & & I & -I \\ & & & & & & & I \end{bmatrix}$$

Sweep #2

$$C_2 = \begin{bmatrix} I & & -I & & & & & \\ & I & & & & & & \\ & & I & & & & & \\ & & & I & & & & \\ & & & & I & & -I & \\ & & & & & I & & \\ & & & & & & I & \\ & & & & & & & I \end{bmatrix}$$

Sweep #3

$$C_3 = \begin{bmatrix} I & & & -I & & & & \\ & I & & & & & & \\ & & I & & & & & \\ & & & I & & & & \\ & & & & I & & & \\ & & & & & I & & \\ & & & & & & I & \\ & & & & & & & I \end{bmatrix}$$

$$C_1C_2C_3 = \begin{bmatrix} I & -I & -I & & -I & & & \\ & I & & & & & & \\ & & I & -I & & & & \\ & & & I & & & & \\ & & & & I & -I & -I & \\ & & & & & I & & \\ & & & & & & I & -I \\ & & & & & & & I \end{bmatrix}$$

In the second version (Figures 2.3 and 2.4) the need for $\log P$ back-solve sweeps is eliminated by continuously utilizing P processors during the reduction phase. In contrast to the first version where every other block-row and column is skipped, the complete system is now considered at each sweep. As shown in Figure 2.3, the reconditioning transformation creates *upward*-pointing spikes, which can be annihilated concurrently as the downward-pointing spikes are "pushed" to the left.

During the first sweep the compacted system (2.34) is transformed by C_1 in Figure 2.4 which reconditions $(W_{p_i} - V_{p_i}), i = 0, 2, 4, 6$, and then 4 processors are used as explained in (2.35)-(2.36) resulting in the system depicted on the right in *Sweep #1* of Figure 2.3. During the second sweep the system is transformed by C_2 in Figure 2.4, reconditioning $(W_{p_i}^{(1)} - V_{p_i}^{(1)}), i = 1, 5$. As shown in *Sweep #2* of Figure 2.3, this creates 2 upward-pointing spikes of 2 blocks each—one above block-row 1 and the other above block-row 5. The upward-pointing spikes are annihilated simultaneously on 4 processors by subtracting appropriate multiples of block-row 1 or 5, while concurrently the downward-pointing spikes are pushed to the left by subtracting appropriate multiples of block-row 1 or 5 on 4 other processors. Finally, during the third sweep the system is transformed by C_3 in Figure 2.4, reconditioning $W_{p_3}^{(2)} - V_{p_3}^{(2)}$. As shown in *Sweep #3* of Figure 2.3, this creates an upward spike of 4 blocks which is annihilated simultaneously on 4 processors by subtracting appropriate multiples of block-row 3, while concurrently the spike below $W_{p_3}^{(2)} - V_{p_3}^{(2)}$ is pushed to the left by subtracting appropriate multiples of block-row 3 on 4 other processors. In general, this version keeps P processors active during each sweep of the reduction after the first.

The final system is depicted on the right in *Sweep #3* of Figure 2.3—a Λ-shaped Jacobian requiring a single-sweep back-solve once $\tilde{y}_{p_0}$ has been computed. As in Figure 2.2, $\tilde{y}_{p_0}$ and $\tilde{y}_{p_7+1}$ are computed by solving a $2n \times 2n$ system constructed from the first and last block-rows. Finally, the mesh variable transformation is reversed: $Y = C_1 C_2 C_3 \tilde{Y}$ where $C_1 C_2 C_3$ is shown in the last frame of Figure 2.4. This, of course, is a highly parallel operation and requires as few as $\log(\log P)$ vector subtractions.

Chapter 3

Stability of the Algorithms

Wright analyzed the stability of *SLF*-QR (his Structured QR or SQR) and *SLF*-LU (his Structured LU or SLU) in [Wrig 92] and [Wrig 94], respectively. We touch on these topics only briefly in §3.1 and §3.2 of this chapter, referring the reader to Wright's papers for further details. Wright also published some negative results on the stability of *SLF*-LU. We now have additional insight on this topic which we discuss in §3.2.

The focus of this chapter is the third algorithm, RSCALE. In §3.3, we present a detailed analysis of the stability of a variant of RSCALE applicable when the algorithm is used to solve ABD systems arising from the discretization of a model linear problem. We are able to prove that RSCALE is stable on this problem, under certain assumptions. These assumptions in turn point out some possible shortcomings of RSCALE, which we address with a few simple modifications to the prototype algorithm. Several numerical examples are included to demonstrate that the modifications have the desired effect, and also to test the sharpness of some of the error bounds arising in the stability analysis.

3.1 *SLF*-QR

In [Wrig 92], Wright observes that Structured QR is simply QR-factorization applied to a row and column permuted version of the ABD matrix, and uses this fact in his stability analysis of the algorithm. The permutations are not shown explicitly. Below we show a possible structure for these row and column permutations, applicable to the single-partition variant of *SLF*-QR. Similar structures may be shown for other variants of the algorithm.

Consider transforming the ABD matrix

$$\mathcal{J} = \begin{bmatrix} \mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 & & & & \\ & S_1 & T_1 & & & \\ & & S_2 & T_2 & & \\ & & & \ddots & \ddots & \\ & & & & S_{M-1} & T_{M-1} \end{bmatrix}$$

using *SLF*-QR as shown in §2.2.3. Conceptually, the orthogonal transformation $Q_1 \in \mathcal{R}^{2n\times 2n}$ used in the first stage of the reduction may be embedded in the $(M+1)n \times (M+1)n$ identity matrix to form $\tilde{Q}_1 \in \mathcal{R}^{(M+1)n\times(M+1)n}$, and the first stage may be written

$$\tilde{Q}_1\mathcal{J} = \begin{bmatrix} \mathcal{B}_a & & & & & \mathcal{B}_b \\ \boxed{S_0^{(1)}} & \boxed{R_0^{(1)}} & \boxed{T_0^{(1)}} & & & \\ \boxed{S_1^{(1)}} & & \boxed{T_1^{(1)}} & & & \\ & & S_2 & T_2 & & \\ & & & \ddots & \ddots & \\ & & & & S_{M-1} & T_{M-1} \end{bmatrix}$$

Similarly, the first $M-1$ stages of the reduction may be written

$$\tilde{Q}_{M-1}\tilde{Q}_{M-2}\cdots\tilde{Q}_1\mathcal{J} = \begin{bmatrix} \mathcal{B}_a & & & & & & \mathcal{B}_b \\ S_0^{(1)} & R_0^{(1)} & T_0^{(1)} & & & & \\ S_1^{(2)} & & R_1^{(2)} & T_1^{(2)} & & & \\ \vdots & & & \ddots & \ddots & & \\ \boxed{S_{M-2}^{(M-1)}} & & & & \boxed{R_{M-2}^{(M-1)}} & \boxed{T_{M-2}^{(M-1)}} \\ \boxed{S_{M-1}^{(M-1)}} & & & & & \boxed{T_{M-1}^{(M-1)}} \end{bmatrix}$$

The $2n \times 2n$ compacted matrix involving $\mathcal{B}_a, \mathcal{B}_b, S_{M-1}^{(M-1)}$ and $T_{M-1}^{(M-1)}$ extracted during the last stage of the reduction may instead by relocated to the upper-left corner of the ABD matrix by swapping the appropriate block rows and columns:

$$P_{compact}[\tilde{Q}_{M-1}\tilde{Q}_{M-2}\cdots\tilde{Q}_1\mathcal{J}]P_{compact} = \begin{bmatrix} \mathcal{B}_a & \mathcal{B}_b & & & & \\ S_{M-1}^{(M-1)} & T_{M-1}^{(M-1)} & & & & \\ S_0^{(1)} & & R_0^{(1)} & T_0^{(1)} & & \\ \vdots & & & \ddots & \ddots & \\ S_{M-3}^{(M-2)} & & & & R_{M-3}^{(M-2)} & T_{M-3}^{(M-2)} \\ S_{M-2}^{(M-1)} & T_{M-2}^{(M-1)} & & & & R_{M-2}^{(M-1)} \end{bmatrix}$$

where $P_{compact} \in \mathcal{R}^{(M+1)n\times(M+1)n}$ is the permutation matrix that "bubbles" the last block-row (block-column) to the second block-row (block-column) position. (This permutation actually requires M block-row swaps.) The final orthogonal transformation $Q_M \in \mathcal{R}^{2n\times 2n}$ used to reduce the compacted matrix may be embedded as before in $\tilde{Q}_M \in \mathcal{R}^{(M+1)n\times(M+1)n}$, and the complete single-partition *SLF*-QR reduction may be written

$$\tilde{R} = \tilde{Q}_M P_{compact}[\tilde{Q}_{M-1}\tilde{Q}_{M-2}\cdots\tilde{Q}_1\mathcal{J}]P_{compact}.$$

For an embedded matrix Q_j, it can easily be shown that, if Q_j is orthogonal, $\tilde{Q}_j$ is orthogonal. Since $P_{compact}$ is orthogonal, and since the product of orthogonal matrices is itself orthogonal, the overall reduction may be written simply as

$$\tilde{R} = \hat{Q}\mathcal{J}P_{compact}$$

with $\hat{Q} = \tilde{Q}_M P_{compact}\tilde{Q}_{M-1}\tilde{Q}_{M-2}\cdots\tilde{Q}_1$ orthogonal. $\tilde{R}$ is not upper-triangular, but closer examination of its structure

$$\tilde{R} = \hat{Q}\mathcal{J}P_{compact} = \begin{bmatrix} R_a^{(M)} & T_b^{(M)} & & & & \\ & R_{M-1}^{(M)} & & & & \\ S_0^{(1)} & & R_0^{(1)} & T_0^{(1)} & & \\ \vdots & & & \ddots & \ddots & \\ S_{M-3}^{(M-2)} & & & & R_{M-3}^{(M-2)} & T_{M-3}^{(M-2)} \\ S_{M-2}^{(M-1)} & T_{M-2}^{(M-1)} & & & & R_{M-2}^{(M-1)} \end{bmatrix}$$

shows that it can be made upper-triangular by swapping the appropriate block rows and columns:

$$\hat{R} = P_\triangle \tilde{R} P_\triangle = \begin{bmatrix} R_0^{(1)} & T_0^{(1)} & & & S_0^{(1)} & \\ & \ddots & \ddots & & \vdots & \\ & & R_{M-3}^{(M-2)} & T_{M-3}^{(M-2)} & S_{M-3}^{(M-2)} & \\ & & & R_{M-2}^{(M-1)} & S_{M-2}^{(M-1)} & T_{M-2}^{(M-1)} \\ & & & & R_a^{(M)} & T_b^{(M)} \\ & & & & & R_{M-1}^{(M)} \end{bmatrix}$$

where $P_\triangle \in \mathcal{R}^{(M+1)n\times(M+1)n}$ is the permutation matrix that "bubbles" the first two block-rows (block-columns) to the last two block-row (block-column) positions. Therefore, the factorization stage in single-partition *SLF*-QR or Structured QR is indeed a QR-factorization of a row and column permuted version of the ABD matrix:

$$P_\triangle \hat{R} = \hat{Q}\mathcal{J}P_{compact}P_\triangle.$$

($P_{\triangle}$ and $P_{compact}$ could have been combined into a single permutation earlier in this discussion, but we chose to separate them in order to follow the original reduction algorithm as presented in §2.2.3 as closely as possible.)

Similar embeddings and permutations may be used to show that each variant of *SLF*-QR is equivalent to a QR-factorization of a unique row and column permuted version of the original ABD matrix, and hence is stable. We do not pursue this here. The accuracy of *SLF*-QR is not in question. The shortcoming of *SLF*-QR is its comparatively high computational cost relative to the other solvers, not its stability.

3.2 *SLF*-LU

Wright gives a detailed analysis of Structured LU in [Wrig 94], specifying certain conditions that must be satisfied in order to ensure stability of the algorithm. Essentially, at each step of the reduction as shown in (2.10)-(2.12) of §2.2.3, the number of row pivots used during the LU factorization of $[T_i^T S_{i+1}^T]^T$ must match the number of rapidly increasing fundamental solution modes represented in the underlying DE at that step ([Matt 85]). More precisely, the number of *cross-block* pivots must match, where a cross-block pivot is defined as the exchange of rows between two *distinct* block-rows as opposed to the exchange of rows within a single block-row. (For example, the exchange of rows between the block-row containing T_i and the block-row containing S_{i+1} is a cross-block pivot.) We refer to [Wrig 94] for further details.

In [Wrig 93], Wright gives examples of linear systems for which Gaussian elimination with row partial pivoting is unstable. These are ABD linear systems arising from the discretization of well-posed linear BVODEs. The Gaussian elimination algorithm discussed in [Wrig 93] is not exactly Structured LU, as the boundary blocks in the ABD matrix are involved in the reduction from the start of the algorithm. However, the cause of the instability applies to both Structured LU and *SLF*-LU: the pivoting strategy at each stage of the reduction is not sufficient to control the rapidly increasing fundamental solution modes in the underlying DE. Wright conjectures that, although it is easy to construct such problems, they likely do not arise frequently in practice. Some random testing is included to support the conjecture.

We have done similar random testing on the partitioned variant of *SLF*-LU, and found that instability often occurs when solving certain classes of ABD linear system. Moreover, we incorporated the parallel partitioned variant of *SLF*-LU in software for solving nonlinear BVODEs (§4.4), and found several examples for which *SLF*-LU instability adversely affects the behaviour of the code (§4.4.3).

In §3.2.1, we give a short analysis suggesting where to look for problems that could cause difficulty for *SLF*-LU. In §3.2.2, we show the results of random testing on these problems, confirming that *SLF*-LU instability does indeed occur frequently in some cases. The potential for instability in other variants of *SLF*-LU is addressed in §3.2.3.

3.2.1 Where to Look for *SLF*-LU Instability

In general, it is difficult to determine if the pivoting strategy used in the LU factorization is sufficient to control stability, especially when the ABD system arises from the discretization of a nonlinear BVODE. Systems arising from the discretization of linear, constant-coefficient BVODEs are somewhat easier to analyze. In these problems, however, we have found examples where *SLF*-LU exhibits instability even when the correct number of pivots is used. Thus, it seems that even in simple problems, it is not sufficient to simply match the *number* of row pivots to the *number* of increasing solution modes. The position of the pivoted rows is important as well; in particular, the pivots must be cross-block as defined above.

One scenario is certain, though. If *SLF*-LU does not pivot, it cannot possibly control the increasing solution modes. In fact, when *SLF*-LU does not pivot, it is equivalent to compactification (§2.1). With no pivoting, there is no fill-in at $\tilde{T}_i^{(k)}$ in (2.8), and with $\tilde{T}_i^{(k)} \equiv 0$, (2.8) is equivalent to (2.6) other than for the difference in Jacobian form ((1.8) versus (1.9)).

We focus our attention on the no-pivot scenario in this section. To this end, consider the linear, constant-coefficient equation

$$y'(t) = Ay(t) + q(t), \tag{3.1}$$

where $y, q, \in \mathcal{R}^3$, $A \in \mathcal{R}^{3\times 3}$, and let A have the structure

$$A \equiv \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{bmatrix}$$

This structure arises, for example, from the standard technique of converting a single 3^{rd}-order differential equation into system of three 1^{st}-order equations. Assume that (3.1) appears in a well-posed BVODE (we are not concerned with the boundary conditions or interval of integration in this discussion), and assume that we solve the BVODE using a trapezoidal finite-difference discretization over a mesh of equally-spaced subintervals of size h. In this case,

$$S_i = -[I + (h/2)A], \;\; T_i = [I - (h/2)A], \;\; \forall i = 0, \ldots, M-1$$

and in (2.8) we have:

$$\begin{bmatrix} T_i \\ S_{i+1} \end{bmatrix} \equiv \begin{bmatrix} 1 & -h/2 & 0 \\ 0 & 1 & -h/2 \\ -ah/2 & -bh/2 & 1-ch/2 \\ -1 & -h/2 & 0 \\ 0 & -1 & -h/2 \\ -ah/2 & -bh/2 & -1-ch/2 \end{bmatrix}$$

Three stages of Gaussian elimination are required to transform the i-th slice of the ABD matrix as shown in (2.8). No pivoting is required during the first stage if

$$|ah/2| \leq 1 \tag{3.2}$$

and when the first stage is completed without pivoting, we have

$$\begin{bmatrix} T_i' \\ S_{i+1}' \end{bmatrix} \equiv \begin{bmatrix} 1 & -h/2 & 0 \\ 0 & 1 & -h/2 \\ 0 & -bh/2-ah^2/4 & 1-ch/2 \\ 0 & -h & 0 \\ 0 & -1 & -h/2 \\ 0 & -bh/2-ah^2/4 & -1-ch/2 \end{bmatrix}$$

No pivoting is required during the second stage if

$$h \leq 1, \tag{3.3}$$

$$|bh/2+ah^2/4| \leq 1, \tag{3.4}$$

and when the second stage is completed without pivoting, we have

$$\begin{bmatrix} T_i'' \\ S_{i+1}'' \end{bmatrix} \equiv \begin{bmatrix} 1 & -h/2 & 0 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1-ch/2-(bh/2+ah^2/4)h/2 \\ 0 & 0 & -h^2/2 \\ 0 & 0 & -h \\ 0 & 0 & -1-ch/2-(bh/2+ah^2/4)h/2 \end{bmatrix}$$

Finally, no pivoting is required during the third stage if

$$h^2/2 \leq |1-\alpha|, \tag{3.5}$$

$$h \leq |1-\alpha|, \tag{3.6}$$

$$|-1-\alpha| \leq |1-\alpha|, \tag{3.7}$$

where $\alpha = ch/2 + bh^2/4 + ah^3/8$.

Therefore, if h is sufficiently small, (3.2)-(3.6) will hold and no pivoting will occur during any stage of the factorization if

$$\alpha \;=\; ch/2 + bh^2/4 + ah^3/8 \;\leq\; 0.$$

In other words, if h is sufficiently small, and if a, b and c are similar in magnitude, there is a good chance no pivoting will occur during any stage of the factorization if $c < 0$.

This is somewhat surprising. There is a 50% chance that c will be negative if the elements in the bottom row of A are randomly generated in any interval centered at 0. The remaining question is whether the resulting differential equations (3.1) exhibit a strong enough dichotomy for the lack of pivoting to cause stability problems. We investigate this further in §3.2.2.

Finally, we note than when the ABD system arises from the discretization of a linear, constant-coefficient equation, $S_i \equiv S, T_i \equiv T \;\forall i = 0, \ldots, M-1$, and if no pivoting occurs when transforming the i-th slice of the ABD matrix, no pivoting will occur when transforming $(i+1)$-st slice either. The result of (2.8) without pivoting is:

$$\left[\begin{array}{cccc|c} \cdots & \tilde{S}_i & U_i & \cdots & \tilde{\phi}_i \\ \cdots & -S_{i+1}T_i^{-1}S_i & T_{i+1} & \cdots & \tilde{\phi}_{i+1} \end{array}\right] \longleftarrow L^{-1} \times \left[\begin{array}{cccc|c} \cdots & S_i & T_i & \cdots & \phi_i \\ \cdots & & S_{i+1} & T_{i+1} \; \cdots & \phi_{i+1} \end{array}\right] \tag{3.8}$$

When processing the $(i+1)$-st slice, we then factor

$$\left[\begin{array}{c} T_{i+1} \\ S_{i+2} \end{array}\right] \equiv \left[\begin{array}{c} T_i \\ S_{i+1} \end{array}\right] \equiv \left[\begin{array}{c} T \\ S \end{array}\right]$$

In other words, we compute exactly the same factorization and the no-pivot scenario repeats. Thus, when reducing on a single partition, if *SLF*-LU does not pivot when transforming the first block-row pair in the partition, it will never pivot.

3.2.2 Random Tests

The analysis in the previous section can be extended to single equations of higher order, and to systems of higher-order equations. In general, we claim that when the Jacobian $[\partial f/\partial y]$ of the differential equation arises from the standard technique of converting a system of higher-order equations into a system of first-order equations, there is a greater probability of *SLF*-LU failure due to no pivoting if the elements in the dense row or rows of the Jacobian are negative and close in magnitude. (The nonzero in each sparse row is 1; see Figure 4.1 in §4.2.1 for more examples.) We now show the results of several random tests to substantiate this claim.

Each test ABD system arises from the trapezoidal finite difference discretization of a linear, constant-coefficient differential equation. We divide the test DEs into three classes:

Class 1: $[\partial f/\partial y]_{ij} \in (-\lambda, 0] \bigcup [1]\ \forall i, j,$

Class 2: $[\partial f/\partial y]_{ij} \in [0, \lambda]\ \forall i, j,$

Class 3: $[\partial f/\partial y]_{ij} \in (-\lambda, \lambda]\ \forall i, j,$

where λ is a specified constant. If our conjecture is correct, there should be more instances of potential *SLF*-LU instability when factoring ABD matrices arising from the discretization of structured Class 1 problems with λ not too large. (There is a higher probability of the elements being close in magnitude when they are randomly generated in a smaller interval.)

We measure *SLF*-LU stability by monitoring the growth of $\|S_i^{(i+1)}\|$ as the reduction proceeds down the partition in (2.10)-(2.12). We flag the reduction as potentially unstable if

$$\|S_i^{(i+1)}\| \geq 10^5 \times \|S_0^{(1)}\| \tag{3.9}$$

for some $i \geq 1$.

In each set of experiments, we generate and factor several hundred ABD matrices. In order to save on computing time, in most cases we do not proceed with a reduction once it has been flagged as unstable. We only complete the factorization for problems where we analyze the accuracy of the solution (Figures 3.3 and 3.4). As we are interested only in problems with a strong dichotomy in these tests, if a differential equation is generated with a weak dichotomy or with no dichotomy, it is discarded and not counted in the total.

Figure 3.1 shows the results of four sets of experiments on randomly generated structured linear problems, where the Jacobian structure arises from converting two 5^{th}-order equations into a system of ten 1^{st}-order equations. The resulting Jacobian is of order $n = 10$ and is approximately $\rho = 28\%$ nonzero. In each set of experiment, 300 ABD matrices are generated on each of eight meshes. In the two sets of experiments with larger λ ($\lambda = 100$ and $\lambda = 10$), we see few instances of instability. In the two sets of experiments with smaller λ ($\lambda = 5$ and $\lambda = 1$), we see many more instances, especially with $\lambda = 1$. We also note there are more instances of instability when factoring ABD matrices arising from the discretization of structured Class 1 problems. (In fact, nearly all Class 1 problem reductions are flagged as potentially unstable in the set of experiments with $\lambda = 1$.) All of these results support our conjecture above.

Figure 3.1: Effect of Jacobian scale (λ) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 *structured* linear problems with $n = 10$ and $\rho = 28\%$ nonzero, on meshes ranging from $h = 0.2$ to $h = 0.9$. The structure arises when a system of 2 fifth-order equations is converted into a system of 10 first-order equations.

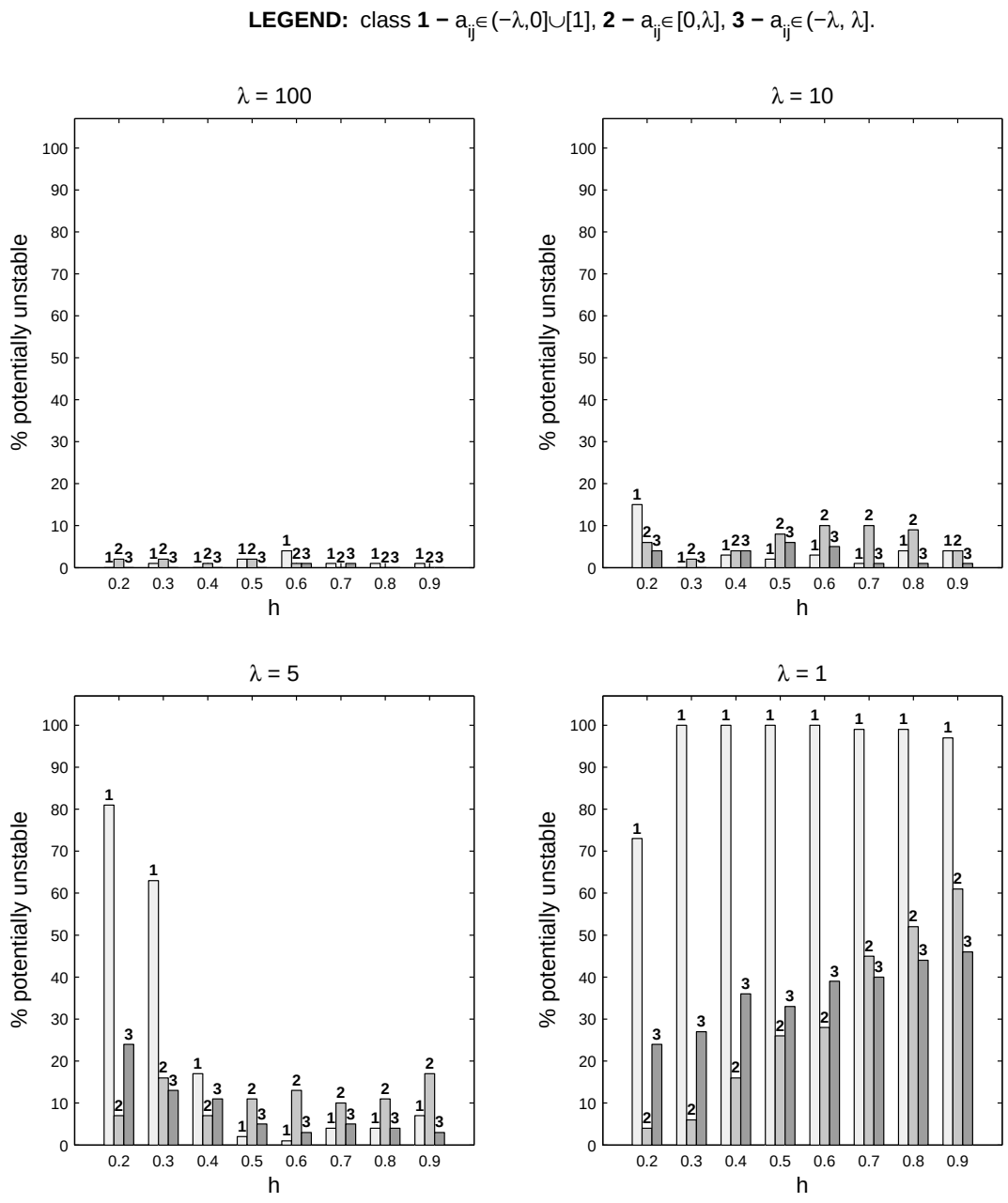


Figure 3.2 shows the results of four sets of experiments similar to those in Figure 3.1, except that the Jacobians are not structured—the nonzeros are randomly distributed. All other experiment parameters, including Jacobian sparsity, are the same as above. Here we see a dramatic drop in the number of potentially unstable *SLF*-LU reductions. This shows the importance of the analysis in §3.2.1. Random testing can sometimes be misleading without some insight into *where* to look.

Figures B.1-B.4 of Appendix B show the results of several other sets of experiments on randomly generated linear problems where the Jacobians are either dense, or sparse with nonzeros randomly distributed. The meshes used in the experiments in Figures B.1 and B.2 are an order of magnitude smaller than in other experiments. The effect of Jacobian order and sparsity on stability is investigated in Figures B.1 and B.2, respectively. The effect of Jacobian scale with dense Jacobians is investigated in Figure B.3, and the effect of Jacobian scale with sparser ($\rho = 50\%$ nonzero) Jacobians is investigated in Figure B.4. At the time of writing, we do not have enough insight into the reasons for the *SLF*-LU instability observed in these experiments to comment further on the results.

Although the measure we use to flag potential instability (3.9) is a reasonably good indicator that something has gone awry in *SLF*-LU, it does not necessarily mean the computed solution to the ABD system will be inaccurate. As a further test, we extract selected Class 3 problems from some of the experiments in Figures 3.1 and 3.2 and solve them to completion using each of the three ABD system solvers (RSCALE, *SLF*-QR and *SLF*-LU). The results are shown in Figures 3.3 and 3.4. The graphs on the right show accuracy statistics for each of the three solvers when used to solve a selected problem. The accuracy of a computed solution is measured as the *algebraic error*—the difference between the computed solution and the solution obtained with a trusted band solver. We also plot the reciprocal condition of the extracted ABD matrix, and the discretization error—the difference between the true solution and the band solver solution. (The problems are constructed in such a way that the true solution is known.) In the context of solving a BVODE, a computed solution is acceptable if its algebraic error is smaller in magnitude than the discretization error. These measures are explained further in Chapter 4 where we discuss the results of many more numerical tests.

The accuracy results in Figures 3.3 and 3.4 clearly show that *SLF*-LU—and only *SLF*-LU—fails to compute an acceptable solution to each problem. Note that each extracted ABD matrix is well-conditioned. Several other examples of *SLF*-LU instability appear in the numerical testing in Chapter 4, when solving both linear and nonlinear BVODEs.

Figure 3.2: Effect of Jacobian scale (λ) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 unstructured linear problems with $n = 10$ and $\rho = 28\%$ nonzero, on meshes ranging from $h = 0.2$ to $h = 0.9$. Jacobian nonzeros are randomly distributed.

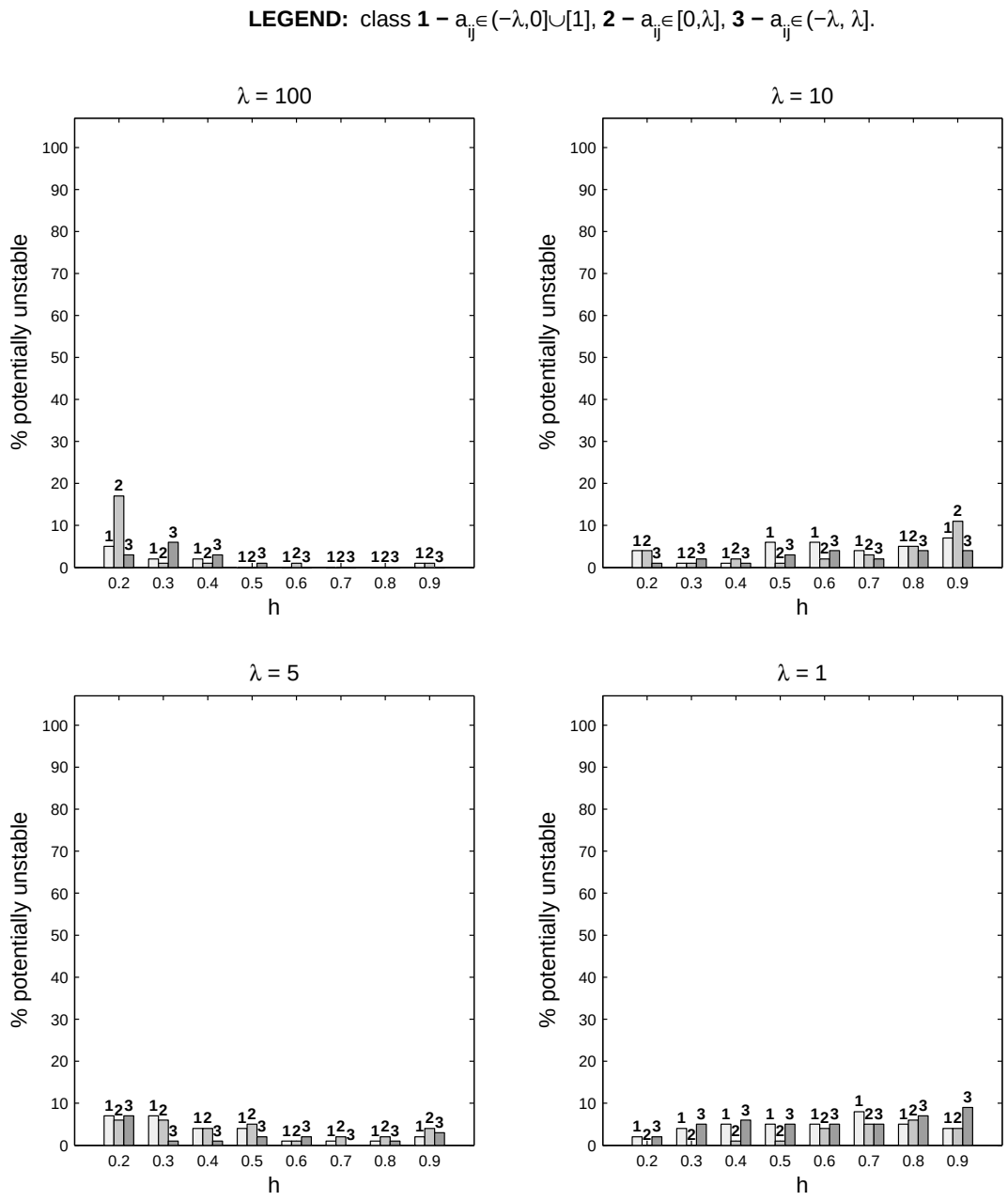


Figure 3.3: Accuracy of *SLF*-QR, *SLF*-LU and RSCALE when used to solve selected class 3 problems from the fourth set of random tests ($\lambda = 1$) in Figure 3.1.

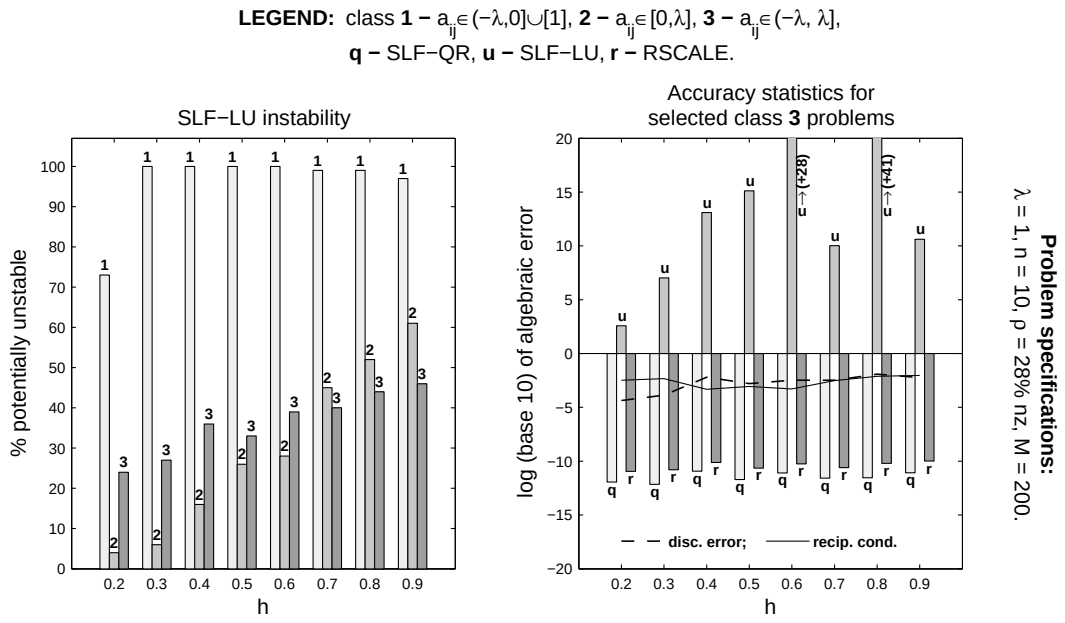
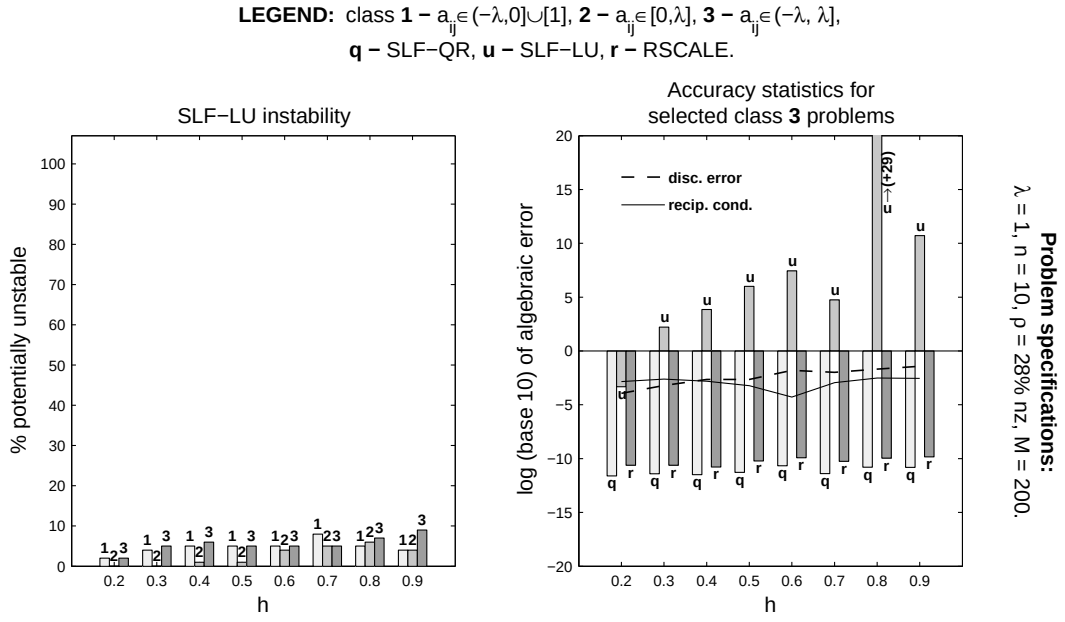


Figure 3.4: Accuracy of *SLF*-QR, *SLF*-LU and RSCALE when used to solve selected class 3 problems from the fourth set of random tests ($\lambda = 1$) in Figure 3.2.



3.2.3 Stability of Other Variants of *SLF*-LU

The analysis, random testing, and accuracy comparisons in §3.2.1 and §3.2.2 all apply to the single-partition variant of *SLF*-LU. In this variant, the ABD matrix is processed in a sequential fashion from top to bottom as shown in (2.10)-(2.12). This algorithm has some features that make it easy to analyze. In particular, if the ABD system arises from the discretization of a constant-coefficient differential equation and the first factorization computed during the reduction does not require pivoting, *SLF*-LU never pivots.

As suggested in §2.2.4, other variants of *SLF*-LU may have different numerical properties. Some experiments in Chapter 4 show instability in a multi-partition variant of *SLF*-LU, where the degree of instability changes as the number of partitions changes. (Surprisingly, the single-partition variant is sometimes stable in these experiments.) Preliminary testing on a cyclic reduction variant of *SLF*-LU shows that most of the problems we have identified in §3.2.1 and §3.2.2 are solved stably using a cyclic reduction approach. This is not surprising, since even in a constant-coefficient problem the blocks S_i, T_i change at each sweep of the reduction regardless of whether pivoting is used (See 2.16-2.19 in §2.2.4.) We leave a more complete analysis of the cyclic reduction variant of *SLF*-LU to future work.

3.3 σ-RSCALE

We now present a stability analysis for a sequential variant of RSCALE described in §2.3, applicable when the algorithm is used to solve ABD systems arising from a constant-stepsize discretization of the model problem

$$y'(t) = Ay(t) + q(t), \quad t \in [t_a, t_b], \quad \mathcal{B}_a y(t_a) + \mathcal{B}_b y(t_b) = g, \tag{3.10}$$

where $y, q, g \in \mathcal{R}^n$, and $A, \mathcal{B}_a, \mathcal{B}_b \in \mathcal{R}^{n\times n}$. We show conditions under which an algebraic equivalent to the rescaling and compactification transformation used in RSCALE does *not* exhibit instability attributable to the growth of rapidly increasing fundamental solution modes in (3.10); instability which is inherent in BlkCR, and which may occur in *SLF*-LU if the pivoting strategy fails.

In §3.3.1 we first give an outline of the variant of RSCALE we analyze, and then derive algebraically equivalent expressions for the various recurrences appearing in the algorithm. These expressions, which are more readily analyzed for stability than the original recurrences, lead to theoretical bounds for the propagation of error at each stage of the algorithm. Our analysis indicates that the prototype algorithm described in §2.3—and included as a FORTRAN

implementation in Appendix E.1—can sometimes fail. We construct some numerical examples in §3.3.2 illustrating this potential instability, and show how adjusting the algorithm as suggested by the analysis corrects the situation. In §3.3.3 we discuss the expected frequency of occurrences of instability. Finally, in order to address some of the shortcomings pointed out by our analysis, in §3.3.4 we suggest a modification to the prototype algorithm which makes it better-suited for variable-coefficient problems.

3.3.1 Stability Analysis

In the analysis that follows we assume little about the underlying constant-stepsize numerical method used to discretize the continuous problem; we require only that the discretization of (3.10) over a mesh of M subintervals results in an ABD system of the form

$$\begin{bmatrix} \mathcal{B}_a & & & & \mathcal{B}_b \\ V_1 & I & & & \\ & V_2 & I & & \\ & & \ddots & \ddots & \\ & & & V_M & I \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} g \\ \phi_1 \\ \phi_2 \\ \vdots \\ \phi_M \end{bmatrix}, \tag{3.11}$$

where $V_k \equiv V \in \mathcal{R}^{n\times n}, \phi_k \in \mathcal{R}^n, k = 1, \ldots, M$, and $y_k \in \mathcal{R}^n, k = 0, \ldots, M$. In fact, the intrinsic characteristics of dichotomy and boundary condition rank, which are closely related to the conditioning of a BVODE, need not be addressed until quite late in the analysis.

The shifted RSCALE algorithm, σ-RSCALE, is briefly introduced in §2.3.2. For ease of notation, $\sigma = 1$ is assumed throughout most of that section, and indeed this choice of σ seems appropriate for most test problems. In this section, however, we show that well-posed BVPs exist for which $\sigma = 1$ is not an appropriate choice. It is therefore necessary to retain σ as a parameter in the discussion that follows.

The steps used to solve (3.11) with σ-RSCALE are listed in Figure 3.5. The system is rescaled in step 1 as shown in (2.23)-(2.27) of §2.3.2, except that here rescaling starts at the second-last, rather than the last, block-row. The rescaled system is compacted in step 2 using a sequential variant of block reduction similar to (2.4) in §2.1. The transformed unknowns are computed in step 3, and the original unknowns are recovered in step 4. Since σ remains constant at each rescaling iteration, we will sometimes refer to the algorithm in Figure 3.5 as *static* σ-RSCALE. A dynamic variant, σ_k-RSCALE, is presented in §3.3.4.

We now proceed toward finding algebraically equivalent expressions for the recurrences

Figure 3.5: Static σ-RSCALE applied to the model ABD system (3.11).

1. **Rescaling.**

   (a) $\hat{W}_{M-1} = [I - \sigma V], \quad \hat{V}_{M-1} = \hat{W}_{M-1}^{-1} V, \quad \hat{\phi}_{M-1} = \hat{W}_{M-1}^{-1} \phi_{M-1}.$

   (b)
   $$\left.\begin{array}{l} \hat{W}_k = [I - \sigma(V - \hat{V}_{k+1})] \\ \hat{V}_k = \hat{W}_k^{-1} V \\ \hat{\phi}_k = \hat{W}_k^{-1}(\phi_k + \sigma \hat{\phi}_{k+1}) \end{array}\right\} \; k = M-2, M-3, \ldots, 1.$$

   (c) $\hat{\mathcal{B}}_a = \mathcal{B}_a(I + \sigma \hat{V}_1), \quad \hat{g} = g + \sigma \mathcal{B}_a \hat{\phi}_1.$

2. **Compactification.**

   (a) $\tilde{V}_1 = \hat{V}_1, \quad \tilde{\phi}_1 = \hat{\phi}_1.$

   (b)
   $$\left.\begin{array}{l} \tilde{V}_k = -\hat{V}_k \tilde{V}_{k-1} \\ \tilde{\phi}_k = \hat{\phi}_k - \hat{V}_k \tilde{\phi}_{k-1} \end{array}\right\} \; k = 2, 3, \ldots, M-1.$$

   (c) $\tilde{V}_M = -V \tilde{V}_{M-1}, \quad \tilde{\phi}_M = \phi_M - V \tilde{\phi}_{M-1}.$

3. **Computation of transformed unknowns.**

   (a) $\tilde{y}_0 = [\hat{\mathcal{B}}_a - \mathcal{B}_b \tilde{V}_M]^{-1} (\hat{g} - \mathcal{B}_b \tilde{\phi}_M).$

   (b) $\tilde{y}_k = \tilde{\phi}_k - \tilde{V}_k \tilde{y}_0, \; k = 1, 2, \ldots, M.$

4. **Recovery of original unknowns.**

   (a) $y_{k-1} = \tilde{y}_{k-1} - \sigma \tilde{y}_k, \; k = 1, 2, \ldots, M-1.$

   (b) $y_{M-1} = \tilde{y}_{M-1}, \quad y_M = \tilde{y}_M.$

appearing in Figure 3.5 which are more readily analyzed for stability. First, we expand the recurrences for $\hat{\phi}_k$ in step 1, $\tilde{\phi}_k$ in step 2, and $\tilde{y}_k$ in step 3 as follows.

Lemma 1 *The recurrence*

$$\hat{\phi}_k = \begin{cases} \hat{W}_k^{-1}\phi_k & k = M-1 \\ \hat{W}_k^{-1}(\phi_k + \sigma\hat{\phi}_{k+1}) & k = M-2, M-3, \ldots, 1 \end{cases}$$

can be written equivalently as

$$\hat{\phi}_k = \sum_{i=k}^{M-1} \sigma^{i-k} \left[\prod_{j=k}^{i} \hat{W}_j^{-1} \right] \phi_i \tag{3.12}$$

for $k = M-1, M-2, \ldots, 1$.

Proof (by induction on k) For $k = M-1$,

$$\begin{aligned} \hat{\phi}_{M-1} &= \hat{W}_{M-1}^{-1}\phi_{M-1} \\ &= \sigma^0 \left[\prod_{j=M-1}^{M-1} \hat{W}_j^{-1} \right] \phi_{M-1} \\ &= \sum_{i=M-1}^{M-1} \sigma^{i-(M-1)} \left[\prod_{j=M-1}^{i} \hat{W}_j^{-1} \right] \phi_i \end{aligned}$$

and (3.12) holds. Assume now that (3.12) holds for an arbitrary $k = s \leq M-1$:

$$\hat{\phi}_s = \sum_{i=s}^{M-1} \sigma^{i-s} \left[\prod_{j=s}^{i} \hat{W}_j^{-1} \right] \phi_i. \tag{3.13}$$

For $k = s-1$, $\hat{\phi}_{s-1} = \hat{W}_{s-1}^{-1}(\phi_{s-1} + \sigma\hat{\phi}_s)$. Substituting (3.13) for $\hat{\phi}_s$,

$$\begin{aligned} \hat{\phi}_{s-1} &= \hat{W}_{s-1}^{-1} \left(\phi_{s-1} + \sigma \left\{ \sum_{i=s}^{M-1} \sigma^{i-s} \left[\prod_{j=s}^{i} \hat{W}_j^{-1} \right] \phi_i \right\} \right) \\ &= \hat{W}_{s-1}^{-1} \left(\phi_{s-1} + \sum_{i=s}^{M-1} \sigma^{i-(s-1)} \left[\prod_{j=s}^{i} \hat{W}_j^{-1} \right] \phi_i \right) \\ &= \hat{W}_{s-1}^{-1}\phi_{s-1} + \sum_{i=s}^{M-1} \sigma^{i-(s-1)} \left[\prod_{j=s-1}^{i} \hat{W}_j^{-1} \right] \phi_i \\ &= \sum_{i=s-1}^{M-1} \sigma^{i-(s-1)} \left[\prod_{j=s-1}^{i} \hat{W}_j^{-1} \right] \phi_i. \end{aligned}$$

Thus (3.12) holds for all $k = M-1, M-2, \ldots, 1$. □

Lemma 2 *The recurrence*

$$\tilde{\phi}_k = \begin{cases} \hat{\phi}_k & k = 1 \\ \hat{\phi}_k - \hat{V}_k \tilde{\phi}_{k-1} & k = 2, 3, \ldots, M-1 \end{cases}$$

can be written equivalently as

$$\tilde{\phi}_k = \hat{\phi}_k + \sum_{i=1}^{k-1} (-1)^{k+i} \left[\prod_{j=1}^{k-i} \hat{V}_{k-j+1} \right] \hat{\phi}_i \tag{3.14}$$

for $k = 1, 2, \ldots, M-1$.

Proof (by induction on k) For $k = 1$,

$$\begin{aligned} \tilde{\phi}_1 &= \hat{\phi}_1 \\ &= \hat{\phi}_1 + \sum_{i=1}^{1-1} (-1)^{1+i} \left[\prod_{j=1}^{1-i} \hat{V}_{1-j+1} \right] \hat{\phi}_i \end{aligned}$$

and (3.14) holds. Assume now that (3.14) holds for an arbitrary $k = s \geq 1$:

$$\tilde{\phi}_s = \hat{\phi}_s + \sum_{i=1}^{s-1} (-1)^{s+i} \left[\prod_{j=1}^{s-i} \hat{V}_{s-j+1} \right] \hat{\phi}_i. \tag{3.15}$$

For $k = s+1$, $\tilde{\phi}_{s+1} = \hat{\phi}_{s+1} - \hat{V}_{s+1} \tilde{\phi}_s$. Substituting (3.15) for $\tilde{\phi}_s$,

$$\begin{aligned} \tilde{\phi}_{s+1} &= \hat{\phi}_{s+1} - \hat{V}_{s+1} \left\{ \hat{\phi}_s + \sum_{i=1}^{s-1} (-1)^{s+i} \left[\prod_{j=1}^{s-i} \hat{V}_{s-j+1} \right] \hat{\phi}_i \right\} \\ &= \hat{\phi}_{s+1} - \hat{V}_{s+1} \hat{\phi}_s + \sum_{i=1}^{s-1} (-1)^{(s+1)+i} \hat{V}_{s+1} \left[\prod_{j=1}^{s-i} \hat{V}_{s-j+1} \right] \hat{\phi}_i \\ &= \hat{\phi}_{s+1} - \hat{V}_{s+1} \hat{\phi}_s + \sum_{i=1}^{s-1} (-1)^{(s+1)+i} \left[\prod_{j=1}^{(s+1)-i} \hat{V}_{(s+1)-j+1} \right] \hat{\phi}_i \\ &= \hat{\phi}_{s+1} + \sum_{i=1}^{s} (-1)^{(s+1)+i} \left[\prod_{j=1}^{(s+1)-i} \hat{V}_{(s+1)-j+1} \right] \hat{\phi}_i. \end{aligned}$$

Thus (3.14) holds for all $k = 1, 2, \ldots, M-1$. □

Lemma 3 *The recurrence*

$$\tilde{y}_k = \tilde{\phi}_k - \tilde{V}_k \tilde{y}_0 \quad k = 1, 2, \ldots, M-1$$

can be written equivalently as

$$\tilde{y}_k = \tilde{\phi}_k - (-1)^{k+1} \left[\prod_{j=1}^{k} \hat{V}_{k-j+1} \right] \tilde{y}_0 \tag{3.16}$$

for $k = 1, 2, \ldots, M-1$.

Proof (by induction on k) We first show that

$$\tilde{V}_k = (-1)^{k+1} \left[\prod_{j=1}^{k} \hat{V}_{k-j+1} \right] \tag{3.17}$$

for $k = 1, 2, \ldots, M-1$; (3.16) follows immediately from (3.17). For $k = 1$,

$$\begin{aligned} \tilde{V}_1 &= \hat{V}_1 \\ &= (-1)^{1+1} \left[\prod_{j=1}^{1} \hat{V}_{1-j+1} \right] \end{aligned}$$

and (3.17) holds. Assume now that (3.17) holds for an arbitrary $k = s \geq 1$:

$$\tilde{V}_s = (-1)^{s+1} \left[\prod_{j=1}^{s} \hat{V}_{s-j+1} \right]. \tag{3.18}$$

For $k = s+1$, $\tilde{V}_{s+1} = -\hat{V}_{s+1}\tilde{V}_s$. Substituting (3.18) for $\tilde{V}_s$,

$$\begin{aligned} \tilde{V}_{s+1} &= -\hat{V}_{s+1} \left\{ (-1)^{s+1} \left[\prod_{j=1}^{s} \hat{V}_{s-j+1} \right] \right\} \\ &= (-1)^{(s+1)+1} \left[\prod_{j=1}^{s+1} \hat{V}_{(s+1)-j+1} \right]. \end{aligned}$$

Thus (3.17) and (3.16) both hold for all $k = 1, 2, \ldots, M-1$. □

Next, we derive three identities that will allow us to express the matrix products appearing in (3.12), (3.14) and (3.16) explicitly in terms of V and σ:

Lemma 4 *The recurrence*

$$\hat{V}_k = \begin{cases} [I - \sigma V]^{-1} V & k = M-1 \\ [I - \sigma(V - \hat{V}_{k+1})]^{-1} V & k = M-2, M-3, \ldots, 1 \end{cases}$$

can be written equivalently as

$$\hat{V}_k = [I - (-\sigma V)^{M-k+1}]^{-1} [V + \sigma^{M-k}(-V)^{M-k+1}] \tag{3.19}$$

for $k = M-1, M-2, \ldots, 1$.

Proof (by induction on k) For $k = M-1$, $\hat{V}_{M-1} = [I-\sigma V]^{-1}V$. If $[I+\sigma V]$ is non-singular (the issue of singularity in (3.19) is addressed later), then

$$\begin{aligned}\hat{V}_{M-1} &= [I-\sigma V]^{-1}[I+\sigma V]^{-1}[I+\sigma V]V \\ &= ([I+\sigma V][I-\sigma V])^{-1}[V+\sigma V^2] \\ &= [I-(\sigma V)^2]^{-1}[V+\sigma V^2]\end{aligned}$$

and (3.19) holds. Assume now that (3.19) holds for an arbitrary $k = i \leq M-1$:

$$\hat{V}_i = [I-(-\sigma V)^{M-i+1}]^{-1}[V+\sigma^{M-i}(-V)^{M-i+1}]. \tag{3.20}$$

For $k = i-1$, $\hat{V}_{i-1} = [I-\sigma(V-\hat{V}_i)]^{-1}V$. Substituting (3.20) for $\hat{V}_i$,

$$\begin{aligned}\hat{V}_{i-1} &= [I-\sigma(V-[I-(-\sigma V)^{M-i+1}]^{-1}[V+\sigma^{M-i}(-V)^{M-i+1}])]^{-1}V \\ &= ([I-(-\sigma V)^{M-i+1}]^{-1}\{[I-(-\sigma V)^{M-i+1}] - \\ &\quad \sigma[I-(-\sigma V)^{M-i+1}]V+\sigma[V+\sigma^{M-i}(-V)^{M-i+1}]\})^{-1}V \\ &= ([I-(-\sigma V)^{M-i+1}]^{-1}\{I-(-\sigma V)^{M-i+2}\})^{-1}V \\ &= [I-(-\sigma V)^{M-i+2}]^{-1}[V+\sigma^{M-i+1}(-V)^{M-i+2}].\end{aligned}$$

Thus (3.19) holds for all $k = M-1, M-2, \ldots, 1$. □

Lemma 5 *$\tilde{V}_{k,i} \equiv \prod_{j=1}^{k-i}\hat{V}_{k-j+1}$ can be written equivalently as*

$$\tilde{V}_{k,i} = [I-(-\sigma V)^{M-i}]^{-1}[V^{k-i}-(-\sigma)^{M-k}V^{M-i}] \tag{3.21}$$

for $1 \leq k \leq M-1,\ 0 \leq i \leq k-1$.

Proof (by induction on i) For $i = k-1$, $\tilde{V}_{k,k-1} = \hat{V}_k$. Referring to (3.19), clearly (3.21) holds. Assume now that (3.21) holds for an arbitrary $i = s \leq k-1$:

$$\tilde{V}_{k,s} = [I-(-\sigma V)^{M-s}]^{-1}[V^{k-s}-(-\sigma)^{M-k}V^{M-s}]. \tag{3.22}$$

For $i = s-1$,

$$\tilde{V}_{k,s-1} = \prod_{j=1}^{k-s+1}\hat{V}_{k-j+1} = [\prod_{j=1}^{k-s}\hat{V}_{k-j+1}]\hat{V}_s = \tilde{V}_{k,s}\hat{V}_s.$$

Substituting (3.22) for $\tilde{V}_{k,s}$ and (3.19) for $\hat{V}_s$, we have

$$\begin{aligned}\tilde{V}_{k,s-1} &= [I-(-\sigma V)^{M-s}]^{-1}[V^{k-s}-(-\sigma)^{M-k}V^{M-s}] \\ &\quad \times [I-(-\sigma V)^{M-s+1}]^{-1}[V+\sigma^{M-s}(-V)^{M-s+1}].\end{aligned}$$

Assuming that V is diagonalizable, the four matrices enclosed by square braces in the above expression commute. After two interchanges,

$$\begin{aligned}\tilde{V}_{k,s-1} &= [I-(-\sigma V)^{M-s+1}]^{-1}[I-(-\sigma V)^{M-s}]^{-1} \\ &\quad \times [V^{k-s}-(-\sigma)^{M-k}V^{M-s}][V+\sigma^{M-s}(-V)^{M-s+1}].\end{aligned}$$

Simplifying,

$$\begin{aligned}\tilde{V}_{k,s-1} &= [I-(-\sigma V)^{M-s+1}]^{-1}[I-(-\sigma V)^{M-s}]^{-1}[V^{k-s+1}-(-\sigma)^{M-k}V^{M-s+1} \\ &\quad + V^{k-s}\sigma^{M-s}(-V)^{M-s+1}-(-\sigma)^{M-k}V^{M-s}\sigma^{M-s}(-V)^{M-s+1}] \\ &= [I-(-\sigma V)^{M-s+1}]^{-1}[I-(-\sigma V)^{M-s}]^{-1}[V^{k-s+1}-(-\sigma)^{M-k}V^{M-s+1} \\ &\quad - (-\sigma V)^{M-s}V^{k-s+1}+(-\sigma V)^{M-s}(-\sigma)^{M-k}V^{M-s+1}] \\ &= [I-(-\sigma V)^{M-s+1}]^{-1}[I-(-\sigma V)^{M-s}]^{-1} \\ &\quad \times [I-(-\sigma V)^{M-s}][V^{k-s+1}-(-\sigma)^{M-k}V^{M-s+1}] \\ &= [I-(-\sigma V)^{M-s+1}]^{-1}[V^{k-s+1}-(-\sigma)^{M-k}V^{M-s+1}].\end{aligned}$$

Thus (3.21) holds for all $i = k-1, k-2, \ldots, 0$. □

Lemma 6 *$\tilde{W}_{k,i} \equiv \prod_{j=k}^{i}\hat{W}_j^{-1}$ can be written equivalently as*

$$\tilde{W}_{k,i} = [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-i}] \tag{3.23}$$

for $1 \le k \le M-1,\ k \le i \le M-1$.

Proof (by induction on i) For $i = k$, $\tilde{W}_{k,k} = \hat{W}_k^{-1}$. Referring to (3.19),

$$\begin{aligned}\hat{W}_k^{-1} &= \hat{V}_k V^{-1} \\ &= [I-(-\sigma V)^{M-k+1}]^{-1}[V+\sigma^{M-k}(-V)^{M-k+1}]V^{-1} \\ &= [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-k}]\end{aligned}$$

and (3.23) holds. Assume now that (3.23) holds for an arbitrary $i = s \ge k$:

$$\tilde{W}_{k,s} = [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-s}]. \tag{3.24}$$

For $i = s+1$,

$$\tilde{W}_{k,s+1} = \prod_{j=k}^{s+1}\hat{W}_j^{-1} = [\prod_{j=k}^{s}\hat{W}_j^{-1}]\hat{W}_{s+1}^{-1} = \tilde{W}_{k,s}\hat{V}_{s+1}V^{-1}.$$

Substituting (3.24) for $\tilde{W}_{k,s}$ and (3.19) for $\hat{V}_{s+1}$, we have

$$\begin{aligned}\tilde{W}_{k,s+1} &= [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-s}] \\ &\quad \times [I-(-\sigma V)^{M-s}]^{-1}[V+\sigma^{M-s-1}(-V)^{M-s}]V^{-1} \\ &= [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-s-1}].\end{aligned}$$

Thus (3.23) holds for all $i = k, k+1, \ldots, M-1$. □

Considering (3.12) and Lemmas 4-6, there appears to be at least three potential sources of instability in σ-RSCALE; namely, for some k, $1 \le k \le M$,

- σ^k grows too large or too small, and/or
- $[I-(-\sigma V)^k]$ is exactly or nearly singular.

If σ^k grows too large, there is potential for instability in the computation of (3.12). If σ^k grows too small, $\hat{V}_k \approxeq V$ and $\tilde{V}_{k,i} \approxeq V^{k-i}$, i.e. σ-RSCALE suffers from the same instability as BlkCR (note that if $\sigma = 0$, the algorithms are identical). The identities derived in Lemmas 4-6 clearly show why $[I-(-\sigma V)^k]$ must remain non-singular. In order to address these instabilities in our analysis, we require that the following two criterion are met by σ.

Criterion 1 *$\sigma \in \mathcal{R}$, $\sigma > 0$ and $\exists\, \Omega_\sigma < \infty$ such that*

$$\begin{cases} 1 \le \sigma^k \le \Omega_\sigma & \sigma \ge 1 \\ 1/\Omega_\sigma \le \sigma^k < 1 & \sigma < 1 \end{cases}$$

for all k, $1 \le k \le M$. Clearly,

$$\omega_\sigma = \begin{cases} \sigma^M & \sigma \ge 1 \\ 1/\sigma^M & \sigma < 1 \end{cases}$$

is the smallest such Ω_σ.

Criterion 2 *$\exists\, \Omega_\lambda > 0$ such that $0 < \Omega_\lambda \le |1-|\lambda_j||$ for all eigenvalues $\lambda_j = \sigma\nu_j$ of (σV). Clearly, $\omega_\lambda = \min_j |1-|\lambda_j||$ is the largest such Ω_λ.*

Exact or near singularity in $[I-(-\sigma V)^k]$ is thus avoided by selecting σ to shift the eigenvalues of V in such a way that ω_λ is sufficiently greater than zero. Numerical examples presented in §3.3.2 illustrate that when σ-RSCALE is used to solve an ABD system of reasonable size, usually it is possible to choose σ so that ω_λ is not too small while at the same time ω_σ is not too large, and often $\sigma = 1$ is an appropriate choice. See §3.3.2 for further details.

In the proof of Lemma 5 we assumed that V is diagonalizable. This property is also required in the analysis that follows; we now state it formerly:

Criterion 3 *$V \in \mathcal{R}^{n\times n}$ is diagonalizable (nondefective); i.e. $\exists$ a nonsingular $S \in \mathcal{C}^{n\times n}$ such that $V = S\,\mathrm{diag}\{\nu_1, \ldots, \nu_n\}S^{-1}$, where $\nu_j \in \mathcal{C}$, $j = 1, \ldots, n$.*

We now derive bounds for $\|\tilde{V}_{k,i}\|_2$ in Lemma 5 and $\|\tilde{W}_{k,i}\|_2$ in Lemma 6.

Lemma 7 *If ω_σ, ω_λ and S are defined as stated in Criteria 1-3, then*

$$\|\tilde{V}_{k,i}\|_2 \;\le\; \begin{cases} \mathcal{K}_2(S)\,[(2+\omega_\lambda)/\omega_\lambda] & \sigma \ge 1 \\ \mathcal{K}_2(S)\,[(2+\omega_\lambda)(\omega_\sigma/\omega_\lambda)] & \sigma < 1 \end{cases} \tag{3.25}$$

for $1 \le k \le M-1,\ 0 \le i \le k-1$, where $\mathcal{K}_2(S) = \|S\|_2\,\|S^{-1}\|_2$.

Proof From Lemma 5, and noting that $\sigma \neq 0$ (Criterion 1),

$$\begin{aligned} \tilde{V}_{k,i} &= [I - (-\sigma V)^{M-i}]^{-1}[V^{k-i} - (-\sigma)^{M-k}V^{M-i}] \\ &= [I - (-\sigma V)^{M-i}]^{-1}[-\sigma^{i-k}\{(-\sigma V)^{k-i} - (-\sigma V)^{M-i}\}]. \end{aligned}$$

Let $\alpha_j, j = 1, \ldots, n$ represent the eigenvalues of $\tilde{V}_{k,i}$. Given that S diagonalizes V, clearly it also diagonalizes $\tilde{V}_{k,i}$. Therefore,

$$\alpha_j = -\sigma^{i-k}\{(-\lambda_j)^{k-i} - (-\lambda_j)^{M-i}\}/(1 - (-\lambda_j)^{M-i})$$

and

$$|\alpha_j| \le \sigma^{i-k}\{|\lambda_j|^{k-i} + |\lambda_j|^{M-i}\}/|1 - |\lambda_j|^{M-i}| \tag{3.26}$$

where $\lambda_j = \sigma\nu_j$ is the j-th eigenvalue of (σV). We now derive bounds for $|\alpha_j|$ in terms of ω_σ and ω_λ which cover all possible λ_j. Given Criterion 2, two cases need be considered:

1. $|\lambda_j| < 1 \Rightarrow |\lambda_j| \le 1 - \omega_\lambda$. (This also implies that $\omega_\lambda \in (0, 1]$.) Substituting into (3.26), and noting that $M - i > k - i > 0$,

$$\begin{aligned} |\alpha_j| &\le \sigma^{i-k}\{(1-\omega_\lambda) + (1-\omega_\lambda)\}/(1 - (1-\omega_\lambda)) \\ &= 2\sigma^{i-k}(1-\omega_\lambda)/\omega_\lambda. \end{aligned}$$

2. $|\lambda_j| > 1 \Rightarrow |\lambda_j| \ge 1 + \omega_\lambda$. Since $|\lambda_j| \neq 0$, we can divide numerator and denominator in (3.26) through by $|\lambda_j|^{M-i}$ giving

$$|\alpha_j| \le \sigma^{i-k}\{1/|\lambda_j|^{M-k} + 1\}/|1/|\lambda_j|^{M-i} - 1|. \tag{3.27}$$

Substituting $|\lambda_j| \ge 1 + \omega_\lambda$ into (3.27), and noting that $M - i > M - k > 0$,

$$\begin{aligned} |\alpha_j| &\le \sigma^{i-k}\{1/(1+\omega_\lambda) + 1\}/|1/(1+\omega_\lambda) - 1| \\ &= \sigma^{i-k}(2+\omega_\lambda)/\omega_\lambda. \end{aligned}$$

Combining the two cases, $|\alpha_j| \leq \sigma^{i-k}(2+\omega_\lambda)/\omega_\lambda$. Now,

$$\begin{aligned} \|\tilde{V}_{k,i}\|_2 &= \|S\,\mathrm{diag}\{\alpha_1,\ldots,\alpha_n\}\,S^{-1}\|_2 \\ &\leq \|S\|_2\,\|\mathrm{diag}\{\alpha_1,\ldots,\alpha_n\}\|_2\,\|S^{-1}\|_2 \\ &= \mathcal{K}_2(S)\,\|\mathrm{diag}\{\alpha_1,\ldots,\alpha_n\}\|_2 \end{aligned}$$

and since $\|\mathrm{diag}\{\alpha_1,\ldots,\alpha_n\}\|_2 \equiv \max_j |\alpha_j|$,

$$\|\tilde{V}_{k,i}\|_2 \leq \mathcal{K}_2(S)\,[\sigma^{i-k}(2+\omega_\lambda)/\omega_\lambda]. \tag{3.28}$$

Finally, $\sigma^{i-k} = 1/\sigma^{k-i}$ with $1 \leq k-i < M$, and, given Criterion 1,

$$1/\sigma^{k-i} \leq \begin{cases} 1 & \sigma \geq 1 \\ \omega_\sigma & \sigma < 1 \end{cases} \tag{3.29}$$

Substituting (3.29) for σ^{i-k} in (3.28) gives (3.25). □

Lemma 8 *If ω_λ and S are defined as stated in Criteria 2-3, then*

$$\|\tilde{W}_{k,i}\|_2 \;\leq\; \mathcal{K}_2(S)\,[(2+\omega_\lambda)/\omega_\lambda] \tag{3.30}$$

for $1 \leq k \leq M-1,\ k \leq i \leq M-1$.

Proof From Lemma 6,

$$\tilde{W}_{k,i} = [I-(-\sigma V)^{M-k+1}]^{-1}[I-(-\sigma V)^{M-i}].$$

Let $\beta_j, j = 1,\ldots,n$ represent the eigenvalues of $\tilde{W}_{k,i}$. Given that S diagonalizes V, clearly it also diagonalizes $\tilde{W}_{k,i}$. Therefore,

$$\beta_j = (1-(-\lambda_j)^{M-i})/(1-(-\lambda_j)^{M-k+1})$$

and

$$|\beta_j| \leq (1+|\lambda_j|^{M-i})/|1-|\lambda_j|^{M-k+1}| \tag{3.31}$$

where $\lambda_j = \sigma\nu_j$ is the j-th eigenvalue of (σV). We now derive bounds for $|\beta_j|$ in terms of ω_λ which cover all possible λ_j. Given Criterion 2, two cases need be considered:

1. $|\lambda_j| < 1 \Rightarrow |\lambda_j| \leq 1-\omega_\lambda$. (This also implies that $\omega_\lambda \in (0,1]$.) Substituting into (3.31), and noting that $M-k+1 > M-i > 0$,

$$\begin{aligned} |\beta_j| &\leq (1+(1-\omega_\lambda))/(1-(1-\omega_\lambda)) \\ &= (2-\omega_\lambda)/\omega_\lambda. \end{aligned}$$

2. $|\lambda_j| > 1 \Rightarrow |\lambda_j| \geq 1 + \omega_\lambda$. Since $|\lambda_j| \neq 0$, we can divide numerator and denominator in (3.31) through by $|\lambda_j|^{M-i}$ giving

$$|\beta_j| \leq (1/|\lambda_j|^{M-i} + 1)/|1/|\lambda_j|^{M-i} - |\lambda_j|^{i-k+1}|. \tag{3.32}$$

Substituting $|\lambda_j| \geq 1 + \omega_\lambda$ into (3.32), and noting that $M - i > 0$ and $i - k + 1 > 0$,

$$\begin{aligned} |\beta_j| &\leq (1/(1+\omega_\lambda) + 1)/|1/(1+\omega_\lambda) - (1+\omega_\lambda)| \\ &= (2+\omega_\lambda)/[(1+\omega_\lambda)^2 - 1] \\ &= (2+\omega_\lambda)/(2\omega_\lambda + \omega_\lambda^2) \\ &= 1/\omega_\lambda. \end{aligned}$$

Combining the two cases, $|\beta_j| \leq (2+\omega_\lambda)/\omega_\lambda$ with $\omega_\lambda \in (0, \infty)$. Now,

$$\begin{aligned} \|\tilde{W}_{k,i}\|_2 &= \|S \,\mathrm{diag}\{\beta_1, \ldots, \beta_n\}\, S^{-1}\|_2 \\ &\leq \|S\|_2 \,\|\mathrm{diag}\{\beta_1, \ldots, \beta_n\}\|_2 \,\|S^{-1}\|_2 \\ &= \mathcal{K}_2(S)\, \|\mathrm{diag}\{\beta_1, \ldots, \beta_n\}\|_2 \end{aligned}$$

and since $\|\mathrm{diag}\{\beta_1, \ldots, \beta_n\}\|_2 \equiv \max_j |\beta_j|$, (3.30) holds. □

Lemma 7 illustrates a key strength of σ-RSCALE. With a suitable σ, $\|\tilde{V}_{k,i}\|_2$ can be made sufficiently small thus preventing excessive block growth during compactification, which is the primary reason both BlkCR and *SLF*-LU sometimes fail on problems with a strong dichotomy. In fact, $\|\tilde{V}_{k,i}\|_2$ can be made *arbitrarily* small with σ sufficiently large. (Note that the bound for $\sigma \geq 1$ is conservative when σ is large.) Unfortunately, however, this property of σ-RSCALE rarely can be exploited since other stages of the algorithm—in particular the computation of $\hat{\phi}_k$ in (3.12)—restrict the size of σ.

Using the bounds derived in Lemmas 7 and 8, we can now begin to analyze the propagation of error in the computation of $\hat{\phi}_k$ in step 1(b), $\tilde{\phi}_k$ in step 2(b) and $\tilde{y}_k$ in step 3(b) of σ-RSCALE.

Lemma 9 *Let $\bar{\Phi} = [\bar{g}^T, \bar{\phi_1}^T, \ldots, \bar{\phi_M}^T]^T \in \mathcal{R}^{(M+1)n}$ denote the computed (approximate) right-hand-side of (3.11), and let $\eta_{\phi_k} = \|\phi_k - \bar{\phi_k}\|_2$ represent the absolute error in ϕ_k, $1 \leq k \leq M-1$. (η_{ϕ_k} includes both round-off and propagated error.) Let $\overline{\tilde{y}_0}$ denote the computed solution to the compacted system in step 3(a) of σ-RSCALE (Figure 3.5), and let $\eta_{\tilde{y}_0} = \|\tilde{y}_0 - \overline{\tilde{y}_0}\|_2$ represent the absolute error in $\tilde{y}_0$. Let $\overline{\hat{\phi}_k}$, $\overline{\tilde{\phi}_k}$ and $\overline{\tilde{y}_k}$ denote the computed values for $\hat{\phi}_k$ in step 1(b), $\tilde{\phi}_k$ in step 2(b) and $\tilde{y}_k$ in step 3(b) of σ-RSCALE, respectively, and let $\eta_{\hat{\phi}_k} = \|\hat{\phi}_k - \overline{\hat{\phi}_k}\|_2$, $\eta_{\tilde{\phi}_k} = \|\tilde{\phi}_k - \overline{\tilde{\phi}_k}\|_2$ and $\eta_{\tilde{y}_k} = \|\tilde{y}_k - \overline{\tilde{y}_k}\|_2$ represent the absolute error in $\hat{\phi}_k$, $\tilde{\phi}_k$ and $\tilde{y}_k$, respectively. Let $\boldsymbol{\eta_\phi} = [\eta_{\phi_1}, \ldots, \eta_{\phi_{M-1}}]^T \in \mathcal{R}^{M-1}$, and $\boldsymbol{\eta_{\hat{\phi}}} = [\eta_{\hat{\phi}_1}, \ldots, \eta_{\hat{\phi}_{M-1}}]^T \in \mathcal{R}^{M-1}$.*

If ω_σ, ω_λ and S are defined as stated in Criteria 1-3, then the propagation of error through the k-th iteration of steps 1(b), 2(b) and 3(b) of σ-RSCALE is bound as follows:

$$\eta_{\hat{\phi}_k} \leq \omega_\sigma \mathcal{K}_2(S)\left[(2+\omega_\lambda)/\omega_\lambda\right]\|\boldsymbol{\eta_\phi}\|_1 \tag{3.33}$$

$$\eta_{\tilde{\phi}_k} \leq \begin{cases} \eta_{\hat{\phi}_k} + \mathcal{K}_2(S)\left[(2+\omega_\lambda)/\omega_\lambda\right]\|\boldsymbol{\eta_{\hat{\phi}}}\|_1 & |\sigma| \geq 1 \\ \eta_{\hat{\phi}_k} + \mathcal{K}_2(S)\left[(2+\omega_\lambda)(\omega_\sigma/\omega_\lambda)\right]\|\boldsymbol{\eta_{\hat{\phi}}}\|_1 & |\sigma| < 1 \end{cases} \tag{3.34}$$

$$\eta_{\tilde{y}_k} \leq \begin{cases} \eta_{\tilde{\phi}_k} + \mathcal{K}_2(S)\left[(2+\omega_\lambda)/\omega_\lambda\right]\eta_{\tilde{y}_0} & |\sigma| \geq 1 \\ \eta_{\tilde{\phi}_k} + \mathcal{K}_2(S)\left[(2+\omega_\lambda)(\omega_\sigma/\omega_\lambda)\right]\eta_{\tilde{y}_0} & |\sigma| < 1 \end{cases} \tag{3.35}$$

Proof Writing the recurrences in steps 1(b), 2(b) and 3(b) of σ-RSCALE in the form of expansions (3.12), (3.14) and (3.16), respectively, we may derive (3.33), (3.34) and (3.35) using the absolute error propagation formula for a linear function of several variables; namely, if $x_i \in \mathcal{R}^n, X = [x_1^T, \ldots, x_k^T]^T \in \mathcal{R}^{kn}, \bar{X} \approxeq X$ and $\Gamma : \mathcal{R}^{kn} \to \mathcal{R}^n$, then

$$\|\Gamma(X) - \Gamma(\bar{X})\|_2 \leq \sum_{i=1}^{k} \|\partial\Gamma/\partial x_i\|_2 \|x_i - \bar{x}_i\|_2. \tag{3.36}$$

First, applying (3.36) to (3.12) we have

$$\begin{aligned} \eta_{\hat{\phi}_k} &= \|\hat{\phi}_k - \overline{\hat{\phi}_k}\|_2 \\ &\leq \sum_{i=k}^{M-1} \left\| \sigma^{i-k} \prod_{j=k}^{i} \hat{W}_j^{-1} \right\|_2 \|\phi_i - \bar{\phi}_i\|_2. \end{aligned}$$

Substituting ω_σ for σ^{i-k} (Criterion 1), (3.30) for $\|\prod_{j=k}^{i} \hat{W}_j^{-1}\|_2$ (Lemma 8), and $\|\boldsymbol{\eta_\phi}\|_1$ for $\sum_{i=k}^{M-1} \|(\phi_i - \bar{\phi}_i)\|_2$, gives (3.33). Similarly, applying (3.36) to (3.14) we have

$$\begin{aligned} \eta_{\tilde{\phi}_k} &= \|\tilde{\phi}_k - \overline{\tilde{\phi}_k}\|_2 \\ &\leq \|\hat{\phi}_k - \overline{\hat{\phi}_k}\|_2 + \sum_{i=1}^{k-1} \left\| (-1)^{k+i} \prod_{j=1}^{k-i} \hat{V}_{k-j+1} \right\|_2 \|\hat{\phi}_i - \overline{\hat{\phi}_i}\|_2. \end{aligned}$$

Substituting (3.25) for $\|\prod_{j=1}^{k-i} \hat{V}_{k-j+1}\|_2$ (Lemma 7) and $\|\boldsymbol{\eta_{\hat{\phi}}}\|_1$ for $\sum_{i=1}^{k-1} \|\hat{\phi}_i - \overline{\hat{\phi}_i}\|_2$, gives (3.34). Finally, applying (3.36) to (3.16) we have

$$\begin{aligned} \eta_{\tilde{y}_k} &= \|\tilde{y}_k - \overline{\tilde{y}_k}\|_2 \\ &\leq \|\tilde{\phi}_k - \overline{\tilde{\phi}_k}\|_2 + \left\| (-1)^{k+1} \prod_{j=1}^{k} \hat{V}_{k-j+1} \right\|_2 \|\tilde{y}_0 - \overline{\tilde{y}_0}\|_2. \end{aligned}$$

Substituting (3.25) for $\|\prod_{j=1}^{k} \hat{V}_{k-j+1}\|_2$ (Lemma 7) gives (3.35). □

A suitable combination of (3.33)-(3.35) will enable us to measure how small perturbations in the right-hand-side $\bar{\Phi}$ of (3.11) affect the accuracy of the computed solution $\bar{Y}$. First, however, we must bound $\eta_{\tilde{y}_0}$ in (3.35), and this requires an estimate for the condition number of the compacted matrix $C = [\hat{\mathcal{B}}_a - \mathcal{B}_b \tilde{V}_M]$ arising in step 3(a) of σ-RSCALE. The condition number of C is related to the condition number of the full ABD matrix $\mathcal{J}$ in (3.11), which in turn is related to the system of ODEs, boundary conditions (3.10) and underlying numerical method used to discretize the continuous problem. When a BVODE is well-posed, the left and right boundary conditions sufficiently "control" the decaying and growing fundamental solution modes, respectively, and an ABD matrix $\mathcal{J}$ arising from a reasonable discretization usually is well-conditioned. The extent to which the boundary conditions must control the solution modes depends largely on the rate of decay and growth of the modes (i.e. the strength of the dichotomy) and on the length of the interval of integration. For example, if each solution mode decays or grows only slowly over the entire problem interval, and the interval is short, most likely *any* set of boundary conditions of sufficient rank will lead to a well-posed BVODE. If, on the other hand, one or more solution mode decays or grows rapidly, sufficient rank is no longer enough as the span of the basis of the boundary equations becomes increasingly relevant. (These issues are discussed more fully in [Asch 88].) The characteristics of dichotomy and interval length in the continuous problem are manifest in the ABD matrix $\mathcal{J}$ as the magnitude of the eigenvalues of the variational matrix V and the number of block-rows M, respectively. A precise measurement of how the boundary conditions affect the condition number of $\mathcal{J}$, and hence the condition number of C, would necessitate the derivation of a condition number estimate incorporating, among other parameters, $\mathcal{B}_a$, $\mathcal{B}_b$, $\|V\|$ and M. This is a non-trivial task and is not pursued here. Instead, we choose to derive a bound for the condition number of C by comparing it to that of the compacted matrix arising during *decoupling*—a proven stable sequential algorithm for solving ABD linear systems.

We first outline a simple variant of the decoupling algorithm; other variants and additional details may be found in [Asch 88]. Initially, we assume that the boundary conditions in (3.10) satisfy the following criterion.

Criterion 4 *Let V be diagonalizable (nondefective); i.e. $\exists$ a nonsingular $S \in \mathcal{C}^{n\times n}$ such that $V = S\,\mathrm{diag}\{\nu_1, \ldots, \nu_n\}S^{-1}$, with eigenvalues ν_j, $j = 1, \ldots, n$ appearing in non-decreasing order of magnitude*

$$|\nu_1| \leq \cdots \leq |\nu_{n_a}| < 1 \leq |\nu_{n_a+1}| \leq \cdots \leq |\nu_n|$$

and partition $S^{-1} \in \mathcal{C}^{n\times n}$ *as follows*[1]

$$S^{-1} = \begin{bmatrix} S_a^{-1} \\ S_b^{-1} \end{bmatrix},\ S_a^{-1} \in \mathcal{C}^{n_a\times n},\ S_b^{-1} \in \mathcal{C}^{n_b\times n},$$

where $n_b = n - n_a$*. Then* $\mathcal{B}_a,\ \mathcal{B}_b \in \mathcal{R}^{n\times n}$ *must be separable, of rank* n_a *and* n_b*, respectively, with* $S_a^{-1} \in$ row-span$\{\mathcal{B}_a\}$ *and* $S_b^{-1} \in$ row-span$\{\mathcal{B}_b\}$*; i.e.* $\mathcal{B}_a$ *and* $\mathcal{B}_b$ *are of the form*

$$\mathcal{B}_a = Z \begin{bmatrix} S_a^{-1} \\ 0 \end{bmatrix},\quad \mathcal{B}_b = Z \begin{bmatrix} 0 \\ S_b^{-1} \end{bmatrix}, \tag{3.37}$$

where $Z \in \mathcal{C}^{n\times n}$ *is nonsingular.*

We begin this variant of the algorithm by diagonalizing each $V_k, k = 1, \ldots, M$ in (3.11). Given S as defined in Criterion 4, this may be accomplished by multiplying each block-row (except the first) through by S^{-1} and each block-column through by S, giving

$$\begin{bmatrix} \mathcal{B}_a S & & & & \mathcal{B}_b S \\ D_1 & I & & & \\ & D_2 & I & & \\ & & \ddots & \ddots & \\ & & & D_M & I \end{bmatrix} \begin{bmatrix} \breve{y}_0 \\ \breve{y}_1 \\ \breve{y}_2 \\ \vdots \\ \breve{y}_M \end{bmatrix} = \begin{bmatrix} g \\ \breve{\phi}_1 \\ \breve{\phi}_2 \\ \vdots \\ \breve{\phi}_M \end{bmatrix}, \tag{3.38}$$

where $D_k = \text{diag}\{\nu_1, \ldots, \nu_{n_a}, \nu_{n_a+1}, \ldots, \nu_n\}, k = 1, \ldots, M, \breve{\phi}_k = S^{-1}\phi_k, k = 1, \ldots, M$ and $\breve{y}_k = S^{-1}y_k, k = 0, \ldots, M$. Two points are worth noting here:

- Although this step is obviously parallelizable in the constant-coefficient case, it is inherently sequential in the variable-coefficient case since, in general, the diagonalization of V_{k+1} cannot proceed until the diagonalization of V_k is complete.

- In a practical algorithm, diagonalization is unnecessarily costly. As shown in [Asch 88], decoupling can be implemented just as effectively using triangular factorization.

Now if $\mathcal{B}_a$ and $\mathcal{B}_b$ are of the form shown in (3.37), the top block-row of (3.38) may be written

$$Z \begin{bmatrix} S_a^{-1} \\ 0 \end{bmatrix} S\,\breve{y}_0 + Z \begin{bmatrix} 0 \\ S_b^{-1} \end{bmatrix} S\,\breve{y}_M = g. \tag{3.39}$$

[1] S_a^{-1} and S_b^{-1} denote the first n_a and last n_b rows of S^{-1}, respectively. These two partitioned matrices are neither square nor invertible.

Multiplying (3.39) through by Z^{-1} gives

$$\begin{bmatrix} I_{n_a} \\ 0 \end{bmatrix} \breve{y}_0 + \begin{bmatrix} 0 \\ I_{n_b} \end{bmatrix} \breve{y}_M = \breve{g} \tag{3.40}$$

where $I_{n_a} \in \mathcal{R}^{n_a \times n}$ and $I_{n_b} \in \mathcal{R}^{n_b \times n}$ represent the first n_a and last n_b rows of I, respectively, and $\breve{g} = Z^{-1}g$. Thus, by transforming (3.11) as shown in (3.38)-(3.40), and assuming that Z is nonsingular, the boundary equations now explicitly give

$$\breve{y}_0^{\downarrow} = \begin{bmatrix} \breve{y}_{0_1} \\ \vdots \\ \breve{y}_{0_{n_a}} \end{bmatrix} = \begin{bmatrix} \breve{g}_1 \\ \vdots \\ \breve{g}_{n_a} \end{bmatrix} \in \mathcal{R}^{n_a} \tag{3.41}$$

and

$$\breve{y}_M^{\uparrow} = \begin{bmatrix} \breve{y}_{M_{n_a+1}} \\ \vdots \\ \breve{y}_{M_n} \end{bmatrix} = \begin{bmatrix} \breve{g}_{n_a+1} \\ \vdots \\ \breve{g}_n \end{bmatrix} \in \mathcal{R}^{n_b}. \tag{3.42}$$

The remaining unknowns may be computed by using (3.41) as the initial value in a forward recurrence for the decaying solution modes,

$$\breve{y}_k^{\downarrow} = \breve{\phi}_k^{\downarrow} - \mathcal{D}^{\downarrow}\, \breve{y}_{k-1}^{\downarrow}, \quad k = 1, 2, \ldots, M, \tag{3.43}$$

and (3.42) as the initial value in a backward recurrence for the growing solution modes,

$$\breve{y}_k^{\uparrow} = (\mathcal{D}^{\uparrow})^{-1}[\breve{\phi}_{k+1}^{\uparrow} - \breve{y}_{k+1}^{\uparrow}], \quad k = M-1, M-2, \ldots, 0, \tag{3.44}$$

where

$$\breve{y}_k^{\downarrow} = \begin{bmatrix} \breve{y}_{k_1} \\ \vdots \\ \breve{y}_{k_{n_a}} \end{bmatrix}, \ \breve{\phi}_k^{\downarrow} = \begin{bmatrix} \breve{\phi}_{k_1} \\ \vdots \\ \breve{\phi}_{k_{n_a}} \end{bmatrix} \in \mathcal{R}^{n_a}, \ \ \breve{y}_k^{\uparrow} = \begin{bmatrix} \breve{y}_{k_{n_a+1}} \\ \vdots \\ \breve{y}_{k_n} \end{bmatrix}, \ \breve{\phi}_k^{\uparrow} = \begin{bmatrix} \breve{\phi}_{k_{n_a+1}} \\ \vdots \\ \breve{\phi}_{k_n} \end{bmatrix} \in \mathcal{R}^{n_b},$$

and

$$\mathcal{D}^{\downarrow} = \text{diag}\{\nu_1, \ldots, \nu_{n_a}\} \in \mathcal{R}^{n_a \times n_a}, \ \mathcal{D}^{\uparrow} = \text{diag}\{\nu_{n_a+1}, \ldots, \nu_n\} \in \mathcal{R}^{n_b \times n_b}.$$

Since $\|\mathcal{D}^{\downarrow}\|_2 < 1$ and $\|(\mathcal{D}^{\uparrow})^{-1}\|_2 \leq 1$, recurrences (3.43) and (3.44) are *stable* and *neutrally stable*, respectively. Finally, the original unknowns are recovered with $y_k = S\breve{y}_k, k = 0, \ldots, M$.

Although it is possible to fabricate an ODE system where the boundary conditions must satisfy Criterion 4 in order for problem to be well-posed, this criterion is rarely met in practice. The usual minimum criterion for general non-separated boundary conditions in (3.10) is:

Criterion 5 *Let V be diagonalizable (nondefective); i.e. $\exists$ a nonsingular $S \in \mathcal{C}^{n\times n}$ such that $V = S\,\mathrm{diag}\{\nu_1, \ldots, \nu_n\}S^{-1}$, with eigenvalues $\nu_j, j = 1, \ldots, n$ appearing in non-decreasing order of magnitude:*

$$|\nu_1| \leq \cdots \leq |\nu_{n_a}| < 1 \leq |\nu_{n_a+1}| \leq \cdots \leq |\nu_n|.$$

Then $\mathcal{B}_a,\ \mathcal{B}_b \in \mathcal{R}^{n\times n}$ must be of rank n_a and n_b, respectively, where $n_b = n - n_a$.

When the decoupling algorithm is used to solve an ABD system with boundary conditions satisfying this more general criterion only, it may no longer be possible to compute $\breve{y}_0^{\downarrow}$ and $\breve{y}_M^{\uparrow}$ as shown in (3.39)-(3.42). Instead, initial values for recurrences (3.43) and (3.44) typically are computed by solving the compacted system

$$\begin{bmatrix} \mathcal{B}_a S & \mathcal{B}_b S \\ \mathcal{D}_a & \mathcal{D}_b \end{bmatrix} \begin{bmatrix} \breve{y}_0 \\ \breve{y}_M \end{bmatrix} = \begin{bmatrix} g \\ \gamma \end{bmatrix} \tag{3.45}$$

where $\mathcal{B}_a,\ \mathcal{B}_b$ are as specified in Criterion 5, $\mathcal{D}_a,\ \mathcal{D}_b \in \mathcal{R}^{n\times n}$ are of the form

$$\mathcal{D}_a = \begin{bmatrix} (-1)^{M+1}(\mathcal{D}^{\downarrow})^M & 0 \\ 0 & I \end{bmatrix},\ \mathcal{D}_b = \begin{bmatrix} I & 0 \\ 0 & (-1)^{M+1}(\mathcal{D}^{\uparrow})^{-M} \end{bmatrix}$$

and $\gamma \in \mathcal{R}^n$ is an appropriately transformed subvector of $\breve{\Phi}$.

A closer examination of (3.45) sheds some light on the intricate relationship between the boundary conditions and dichotomy, and how they contribute to the condition number of both the compacted and full ABD matrix. For example, let $[X]_{(i,:)}$, $[X]_{(:,j)}$, and $[X]_{(i,j)}$ denote the i-th row, j-th column, and (i,j)-th element, respectively, of $X \in \mathcal{R}^{n\times n}$, and let ϵ represent machine epsilon. If, in (3.45), $|[\mathcal{D}_a]_{(j,j)}| < \epsilon$ for some j, $1 \leq j \leq n_a$, and one or both of the following conditions hold:

1. $\|[\mathcal{B}_a S]_{(:,j)}\| < \epsilon$,
2. $\|[\mathcal{B}_a S]_{(i,:)}\| < \epsilon$ and $\|[\mathcal{B}_b S]_{(i,:)} - [I]_{(j,:)}\| < \epsilon$ for some i, $1 \leq i \leq n$,

then the compacted matrix is either poorly-conditioned or numerically singular, whereas if $|[\mathcal{D}_a]_{(j,j)}| \gg \epsilon$, neither of these conditions is likely to adversely affect the condition number. The magnitude of each of the first n_a elements on the diagonal of $\mathcal{D}_a$ and last n_b elements on the diagonal of $\mathcal{D}_b$ is uniquely determined by the strength of the dichotomy and length of the interval of integration. For instance, in the above example we can easily have $|[\mathcal{D}_a]_{(j,j)}| \ll \epsilon$ when the j-th fundamental solution mode decreases rapidly over a long interval. It therefore

follows that the contribution of general $\mathcal{B}_a$ and $\mathcal{B}_b$ (i.e. $\mathcal{B}_a$ and $\mathcal{B}_b$ which satisfy Criterion 5 *only*) to the condition number of the compacted matrix depends also on the strength of the dichotomy and length of the interval of integration. If, on the other hand, $\mathcal{B}_a$ and $\mathcal{B}_b$ satisfy Criterion 4, the compacted system (3.45) reduces to (3.41), (3.42),

$$\breve{y}_0^{\uparrow} = \begin{bmatrix} \breve{y}_{0_{n_a+1}} \\ \vdots \\ \breve{y}_{0_n} \end{bmatrix} = \begin{bmatrix} \gamma_{n_a+1} \\ \vdots \\ \gamma_n \end{bmatrix} - (-1)^{M+1}(\mathcal{D}^{\uparrow})^{-M} \begin{bmatrix} \breve{y}_{M_{n_a+1}} \\ \vdots \\ \breve{y}_{M_n} \end{bmatrix} \tag{3.46}$$

and

$$\breve{y}_M^{\downarrow} = \begin{bmatrix} \breve{y}_{M_1} \\ \vdots \\ \breve{y}_{M_{n_a}} \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_{n_a} \end{bmatrix} - (-1)^{M+1}(\mathcal{D}^{\downarrow})^{M} \begin{bmatrix} \breve{y}_{0_1} \\ \vdots \\ \breve{y}_{0_{n_a}} \end{bmatrix}. \tag{3.47}$$

Clearly, in this case, the contribution of $\mathcal{B}_a$ and $\mathcal{B}_b$ to the condition number of the compacted matrix may be measured solely in terms of the condition number of Z. (Recall that Z is inverted when computing $\breve{y}_0^{\downarrow}$ in (3.41) and $\breve{y}_M^{\uparrow}$ in (3.42).)

We close this discussion on the decoupling algorithm by showing that when $\mathcal{B}_a$ and $\mathcal{B}_b$ satisfy Criterion 5, the compacted system (3.45) may be further reduced to an $n \times n$ system involving $\breve{y}_0^{\downarrow}$ and $\breve{y}_M^{\uparrow}$ only. To this end, consider partitioning $\mathcal{B}_a$, $\mathcal{B}_b$ and S as follows:

$$\mathcal{B}_a = \begin{bmatrix} \mathcal{B}_{a_1} \\ \mathcal{B}_{a_2} \end{bmatrix}, \ \mathcal{B}_b = \begin{bmatrix} \mathcal{B}_{b_1} \\ \mathcal{B}_{b_2} \end{bmatrix}, \ S = \begin{bmatrix} S_a & S_b \end{bmatrix},$$

where $\mathcal{B}_{a_1}, \mathcal{B}_{b_1} \in \mathcal{R}^{n_a \times n}, \mathcal{B}_{a_2}, \mathcal{B}_{b_2} \in \mathcal{R}^{n_b \times n}, S_a \in \mathcal{R}^{n \times n_a}$ and $S_b \in \mathcal{R}^{n \times n_b}$. Rewriting (3.45) using this partitioning gives

$$\begin{bmatrix} \mathcal{B}_{a_1}S_a & \mathcal{B}_{a_1}S_b & \mathcal{B}_{b_1}S_a & \mathcal{B}_{b_1}S_b \\ \mathcal{B}_{a_2}S_a & \mathcal{B}_{a_2}S_b & \mathcal{B}_{b_2}S_a & \mathcal{B}_{b_2}S_b \\ (-1)^{M+1}(\mathcal{D}^{\downarrow})^M & 0 & I & 0 \\ 0 & I & 0 & (-1)^{M+1}(\mathcal{D}^{\uparrow})^{-M} \end{bmatrix} \begin{bmatrix} \breve{y}_0^{\downarrow} \\ \breve{y}_0^{\uparrow} \\ \breve{y}_M^{\downarrow} \\ \breve{y}_M^{\uparrow} \end{bmatrix} = \begin{bmatrix} g^{\downarrow} \\ g^{\uparrow} \\ \gamma^{\downarrow} \\ \gamma^{\uparrow} \end{bmatrix} \tag{3.48}$$

where, as before, we denote subvectors consisting of the first n_a and last n_b elements of $x \in \mathcal{R}^n$ as $x^{\downarrow} \in \mathcal{R}^{n_a}$ and $x^{\uparrow} \in \mathcal{R}^{n_b}$, respectively. Now interchanging $\breve{y}_0^{\downarrow}$ and $\breve{y}_M^{\downarrow}$ in (3.48) gives

$$\begin{bmatrix} \mathcal{B}_{b_1}S_a & \mathcal{B}_{a_1}S_b & \mathcal{B}_{a_1}S_a & \mathcal{B}_{b_1}S_b \\ \mathcal{B}_{b_2}S_a & \mathcal{B}_{a_2}S_b & \mathcal{B}_{a_2}S_a & \mathcal{B}_{b_2}S_b \\ I & 0 & (-1)^{M+1}(\mathcal{D}^{\downarrow})^M & 0 \\ 0 & I & 0 & (-1)^{M+1}(\mathcal{D}^{\uparrow})^{-M} \end{bmatrix} \begin{bmatrix} \breve{y}_M^{\downarrow} \\ \breve{y}_0^{\uparrow} \\ \breve{y}_0^{\downarrow} \\ \breve{y}_M^{\uparrow} \end{bmatrix} = \begin{bmatrix} g^{\downarrow} \\ g^{\uparrow} \\ \gamma^{\downarrow} \\ \gamma^{\uparrow} \end{bmatrix} \tag{3.49}$$

With the $n \times n$ identity matrix appearing in the lower-left corner of (3.49), we can clearly eliminate $\breve{y}_0^{\uparrow}$ and $\breve{y}_M^{\downarrow}$ from the first n equations by subtracting an appropriate multiple of the last n equations. This leads to the compacted system

$$\breve{C} \begin{bmatrix} \breve{y}_0^{\downarrow} \\ \breve{y}_M^{\uparrow} \end{bmatrix} = \begin{bmatrix} g^{\downarrow} \\ g^{\uparrow} \end{bmatrix} - \begin{bmatrix} \mathcal{B}_{b_1} S_a & \mathcal{B}_{a_1} S_b \\ \mathcal{B}_{b_2} S_a & \mathcal{B}_{a_2} S_b \end{bmatrix} \begin{bmatrix} \gamma^{\downarrow} \\ \gamma^{\uparrow} \end{bmatrix} \tag{3.50}$$

where

$$\breve{C} = \begin{bmatrix} \mathcal{B}_{a_1} S_a & \mathcal{B}_{b_1} S_b \\ \mathcal{B}_{a_2} S_a & \mathcal{B}_{b_2} S_b \end{bmatrix} - \begin{bmatrix} \mathcal{B}_{b_1} S_a & \mathcal{B}_{a_1} S_b \\ \mathcal{B}_{b_2} S_a & \mathcal{B}_{a_2} S_b \end{bmatrix} \begin{bmatrix} (-1)^{M+1} (\mathcal{D}^{\downarrow})^M & 0 \\ 0 & (-1)^{M+1} (\mathcal{D}^{\uparrow})^{-M} \end{bmatrix}.$$

Once (3.50) is solved giving $\breve{y}_0^{\downarrow}$ and $\breve{y}_M^{\uparrow}$, $\breve{y}_0^{\uparrow}$ and $\breve{y}_M^{\downarrow}$ may be computed either as shown in (3.46) and (3.47), or as the last step of recurrences (3.44) and (3.43), respectively.

In the next lemma we bound the condition number of the compacted matrix arising in σ-RSCALE by comparing it to that of $\breve{C}$. Note that since the variant of decoupling outlined above is theoretically stable, the condition number of $\breve{C}$ is closely related to the conditioning of the BVODE. For example, if the continuous problem is well-conditioned, and the full ABD matrix $\mathcal{J}$ arises from a reasonable discretization, then both $\mathcal{J}$ and $\breve{C}$ are guaranteed to be well-conditioned. A thorough analysis of the connection between the conditioning of the continuous and discrete problems is given in [Asch 88].

We now derive an upper bound for the condition number of the compacted matrix C arising in step 3(a) of σ-RSCALE:

Lemma 10 *If ω_σ, ω_λ, λ_j and S are defined as stated in Criteria 1, 2 and 5, then the 2-norm condition number of the compacted matrix $C = \hat{\mathcal{B}}_a - \mathcal{B}_b \tilde{V}_M$ arising in step 3(a) of σ-RSCALE is bound above by*

$$\mathcal{K}_2(C) < 4\, \mathcal{K}_2(\breve{C})\, \mathcal{K}_2(S)\, (\omega_\sigma/\omega_\lambda^2) \max_j [1 + |\lambda_j|, \omega_\sigma] \tag{3.51}$$

where $\breve{C}$ is the compacted matrix arising in the decoupling algorithm (3.50).

Proof Expanding $\hat{\mathcal{B}}_a$ and $\tilde{V}_M$ as computed in steps 1(c) and 2(c) of Figure 3.5, C can be written equivalently as $\mathcal{B}_a C_a - \mathcal{B}_b C_b$ where

$$C_a = I + \sigma \hat{V}_1$$

and

$$C_b = (-1)^{M+1} V \prod_{j=1}^{M-1} \hat{V}_{M-j}.$$

Substituting (3.19) (Lemma 4) for $\hat{V}_1$ in C_a gives

$$\begin{aligned} C_a &= I + \sigma[I - (-\sigma V)^M]^{-1}[V + \sigma^{M-1}(-V)^M] \\ &= [I - (-\sigma V)^M]^{-1}\{[I - (-\sigma V)^M] + \sigma[V + \sigma^{M-1}(-V)^M]\} \\ &= [I - (-\sigma V)^M]^{-1}[I + \sigma V] \end{aligned}$$

and substituting (3.21) (Lemma 5) for $\tilde{V}_{M-1,0} \equiv \prod_{j=1}^{M-1} \hat{V}_{M-j}$ in C_b gives

$$\begin{aligned} C_b &= (-1)^{M+1} V[I - (-\sigma V)^M]^{-1}[V^{M-1} + \sigma V^M] \\ &= (-1)^{M+1}[I - (-\sigma V)^M]^{-1}[V^M + \sigma V^{M+1}]. \end{aligned}$$

A fundamental relationship between C_a and C_b follows immediately; namely

$$C_b C_a^{-1} = (-1)^{M+1} V^M. \tag{3.52}$$

This relationship is exploited later in the proof.

Let α_j and β_j, $j = 1, \ldots, n$, represent the eigenvalues of C_a and C_b, respectively. Given that S diagonalizes V (Criterion 5), clearly it also diagonalizes both C_a and C_b. Therefore,

$$\begin{aligned} C &= \mathcal{B}_a C_a - \mathcal{B}_b C_b \\ &= [\mathcal{B}_a S\mathcal{D}_a - \mathcal{B}_b S\mathcal{D}_b] S^{-1} \end{aligned}$$

or, multiplying through by S,

$$CS = \mathcal{B}_a S\mathcal{D}_a - \mathcal{B}_b S\mathcal{D}_b \tag{3.53}$$

where $\mathcal{D}_a = \text{diag}\{\alpha_1, \ldots, \alpha_n\}$ and $\mathcal{D}_b = \text{diag}\{\beta_1, \ldots, \beta_n\}$. We now partition (3.53) to reflect the dichotomy specified in Criterion 5. Let

$$\mathcal{B}_a = \begin{bmatrix} \mathcal{B}_{a_1} \\ \mathcal{B}_{a_2} \end{bmatrix}, \ \mathcal{B}_b = \begin{bmatrix} \mathcal{B}_{b_1} \\ \mathcal{B}_{b_2} \end{bmatrix}, \ S = \begin{bmatrix} S_a & S_b \end{bmatrix},$$

where $\mathcal{B}_{a_1}, \mathcal{B}_{b_1} \in \mathcal{R}^{n_a \times n}$, $\mathcal{B}_{a_2}, \mathcal{B}_{b_2} \in \mathcal{R}^{n_b \times n}$, $S_a \in \mathcal{R}^{n \times n_a}$, $S_b \in \mathcal{R}^{n \times n_b}$, and let

$$\mathcal{D}_a = \begin{bmatrix} \mathcal{D}_{a_1} & 0 \\ 0 & \mathcal{D}_{a_2} \end{bmatrix}, \ \mathcal{D}_b = \begin{bmatrix} \mathcal{D}_{b_1} & 0 \\ 0 & \mathcal{D}_{b_2} \end{bmatrix},$$

where $\mathcal{D}_{a_1}, \mathcal{D}_{b_1} \in \mathcal{R}^{n_a \times n_a}$ and $\mathcal{D}_{a_2}, \mathcal{D}_{b_2} \in \mathcal{R}^{n_b \times n_b}$. Then (3.53) may be written

$$CS = \begin{bmatrix} \mathcal{B}_{a_1} S_a & \mathcal{B}_{a_1} S_b \\ \mathcal{B}_{a_2} S_a & \mathcal{B}_{a_2} S_b \end{bmatrix} \begin{bmatrix} \mathcal{D}_{a_1} & 0 \\ 0 & \mathcal{D}_{a_2} \end{bmatrix} - \begin{bmatrix} \mathcal{B}_{b_1} S_a & \mathcal{B}_{b_1} S_b \\ \mathcal{B}_{b_2} S_a & \mathcal{B}_{b_2} S_b \end{bmatrix} \begin{bmatrix} \mathcal{D}_{b_1} & 0 \\ 0 & \mathcal{D}_{b_2} \end{bmatrix}. \tag{3.54}$$

Let

$$X = \begin{bmatrix} \mathcal{D}_{a_1} & 0 \\ 0 & -\mathcal{D}_{b_2} \end{bmatrix} \in \mathcal{R}^{n\times n}. \tag{3.55}$$

Multiplying (3.54) through by X^{-1} and simplifying gives

$$CSX^{-1} = \begin{bmatrix} \mathcal{B}_{a_1}S_a & \mathcal{B}_{b_1}S_b \\ \mathcal{B}_{a_2}S_a & \mathcal{B}_{b_2}S_b \end{bmatrix} - \begin{bmatrix} \mathcal{B}_{b_1}S_a & \mathcal{B}_{a_1}S_b \\ \mathcal{B}_{b_2}S_a & \mathcal{B}_{a_2}S_b \end{bmatrix} \begin{bmatrix} \mathcal{D}_{b_1}\mathcal{D}_{a_1}^{-1} & 0 \\ 0 & \mathcal{D}_{a_2}\mathcal{D}_{b_2}^{-1} \end{bmatrix}. \tag{3.56}$$

Now by diagonalizing (3.52), we have

$$\mathcal{D}_b\mathcal{D}_a^{-1} = (-1)^{M+1}\mathcal{D}^M. \tag{3.57}$$

Taking the first n_a equations of (3.57) gives

$$\mathcal{D}_{b_1}\mathcal{D}_{a_1}^{-1} = (-1)^{M+1}(\mathcal{D}^{\downarrow})^M \tag{3.58}$$

where $\mathcal{D}^{\downarrow} = \text{diag}\{\nu_1, \ldots, \nu_{n_a}\}$. Similarly, inverting both sides of (3.57) and then taking the last n_b equations of the result gives

$$\mathcal{D}_{a_2}\mathcal{D}_{b_2}^{-1} = (-1)^{M+1}(\mathcal{D}^{\uparrow})^{-M} \tag{3.59}$$

where $\mathcal{D}^{\uparrow} = \text{diag}\{\nu_{n_a+1}, \ldots, \nu_n\}$. Substituting (3.58) and (3.59) into (3.56), we see that (3.56) is identical to $\breve{C}$ in (3.50); i.e. when $\mathcal{B}_a$ and $\mathcal{B}_b$ satisfy Criterion 5, the compacted matrix C arising in step 3(a) of σ-RSCALE—when transformed with SX^{-1}—is algebraically equivalent to the compacted matrix $\breve{C}$ arising in the decoupling algorithm. It therefore follows that

$$\mathcal{K}_2(C) \leq \mathcal{K}_2(\breve{C})\mathcal{K}_2(S)\mathcal{K}_2(X). \tag{3.60}$$

We now derive a bound for $\mathcal{K}_2(X)$. From above, we have $C = \mathcal{B}_aC_a - \mathcal{B}_bC_b$ where

$$C_a = [I - (-\sigma V)^M]^{-1}[I + \sigma V] \tag{3.61}$$

and

$$C_b = (-1)^{M+1}[I - (-\sigma V)^M]^{-1}[V^M + \sigma V^{M+1}]. \tag{3.62}$$

Since $\sigma \neq 0$ (Criterion 1), (3.62) may be written equivalently as

$$C_b = (-1)^{M+1}[\sigma^M\{I - (-\sigma V)^M\}]^{-1}[(\sigma V)^M + (\sigma V)^{M+1}]. \tag{3.63}$$

Given that S diagonalizes each of V, C_a and C_b, expressions for the eigenvalues α_j of C_a and β_j of C_b follow directly from (3.61) and (3.63), respectively:

$$\alpha_j = (1 + \lambda_j)/(1 - (-\lambda_j)^M) \tag{3.64}$$

$$\beta_j = (-1)^{M+1}(\lambda_j^M + \lambda_j^{M+1})/(\sigma^M\{1-(-\lambda_j)^M\}) \tag{3.65}$$

for $j = 1, \ldots, n$ where $\lambda_j = \sigma\nu_j$ is the j-th eigenvalue of (σV). From (3.55), we see that a subset of these α_j and β_j make up the diagonal of X; namely

$$X = \text{diag}\{\alpha_1, \ldots, \alpha_{n_a}, -\beta_{n_a+1}, \ldots, -\beta_n\}.$$

We next derive bounds, in terms of ω_σ (Criterion 1) and ω_λ (Criterion 2), for the magnitude of each diagonal element in X. Taking absolute values in (3.64) and (3.65), we have

$$\frac{|1-|\lambda_j||}{1+|\lambda_j|^M} \le |\alpha_j| \le \frac{1+|\lambda_j|}{|1-|\lambda_j|^M|} \tag{3.66}$$

$$\frac{||\lambda_j|^M - |\lambda_j|^{M+1}|}{\sigma^M(1+|\lambda_j|^M)} \le |\beta_j| \le \frac{|\lambda_j|^M + |\lambda_j|^{M+1}}{\sigma^M|1-|\lambda_j|^M|}. \tag{3.67}$$

Given the order imposed on the eigenvalues of V as specified in Criterion 5, it follows that each α_j in X arises from an eigenvalue ν_j of V with $|\nu_j| < 1$, and each β_j in X arises from an eigenvalue ν_j of V with $|\nu_j| \ge 1$. We handle these cases separately below, taking into consideration all possible λ_j arising from different combinations of σ and ν_j.

1. Bounds for $|\alpha_j|, j = 1, \ldots, n_a$. Each α_j is computed from ν_j with $|\nu_j| < 1$.

   (a) $\sigma \le 1 \wedge |\nu_j| < 1 \Rightarrow |\lambda_j| = |\sigma\nu_j| \le 1 - \omega_\lambda$ and

   $$0 \le |\lambda_j|^M \le |\lambda_j| \le 1 - \omega_\lambda < 1. \tag{3.68}$$

   Substituting (3.68) into (3.66),

   $$\frac{|1-(1-\omega_\lambda)|}{1+(1-\omega_\lambda)} \le |\alpha_j| \le \frac{1+(1-\omega_\lambda)}{|1-(1-\omega_\lambda)|}.$$

   Simplifying,

   $$\omega_\lambda/(2-\omega_\lambda) \le |\alpha_j| \le (2-\omega_\lambda)/\omega_\lambda. \tag{3.69}$$

   (b) $\sigma > 1 \wedge |\nu_j| < 1 \Rightarrow |\lambda_j| = |\sigma\nu_j| \le 1 - \omega_\lambda \vee |\lambda_j| = |\sigma\nu_j| \ge 1 + \omega_\lambda$. The former case is handled in 1(a). In the latter case, $1 + \omega_\lambda \le |\lambda_j| < \sigma$ and

   $$1 + \omega_\lambda \le |\lambda_j| < |\lambda_j|^M < \sigma^M = \omega_\sigma. \tag{3.70}$$

   Substituting (3.70) into (3.66),

   $$\frac{|1-(1+\omega_\lambda)|}{1+\omega_\sigma} \le |\alpha_j| \le \frac{1+\sigma}{|1-(1+\omega_\lambda)|}.$$

   Simplifying,

   $$\omega_\lambda/(1+\omega_\sigma) \le |\alpha_j| \le (1+\sigma)/\omega_\lambda. \tag{3.71}$$

2. Bounds for $|\beta_j|, j = n_a + 1, \ldots, n$. Each β_j is computed from ν_j with $|\nu_j| \geq 1$.

   (a) $\sigma \geq 1 \wedge |\nu_j| \geq 1 \Rightarrow |\lambda_j| = |\sigma\nu_j| \geq 1 + \omega_\lambda$ and

   $$1 < 1 + \omega_\lambda \leq |\lambda_j| < |\lambda_j|^M < \infty. \tag{3.72}$$

   Noting that $|\lambda_j|^M \neq 0$ in this case, we divide numerator and denominator of each side of (3.67) through by $|\lambda_j|^M$ giving

   $$\frac{|1 - |\lambda_j||}{\sigma^M(1/|\lambda_j|^M + 1)} \leq |\beta_j| \leq \frac{1 + |\lambda_j|}{\sigma^M|1/|\lambda_j|^M - 1|}. \tag{3.73}$$

   Substituting (3.72) into (3.73),

   $$\frac{|1 - (1 + \omega_\lambda)|}{\sigma^M(1/(1 + \omega_\lambda) + 1)} \leq |\beta_j| \leq \frac{1 + |\lambda_j|}{\sigma^M|1/(1 + \omega_\lambda) - 1|}.$$

   Simplifying,

   $$\frac{\omega_\lambda(1 + \omega_\lambda)}{\omega_\sigma(2 + \omega_\lambda)} \leq |\beta_j| \leq \frac{(1 + |\lambda_j|)(1 + \omega_\lambda)}{\omega_\sigma\omega_\lambda}. \tag{3.74}$$

   (b) $\sigma < 1 \wedge |\nu_j| \geq 1 \Rightarrow |\lambda_j| = |\sigma\nu_j| \geq 1 + \omega_\lambda \vee |\lambda_j| = |\sigma\nu_j| \leq 1 - \omega_\lambda$. The former case is handled in 2(a). In the latter case, $\sigma \leq |\lambda_j| \leq 1 - \omega_\lambda$ and

   $$0 < 1/\omega_\sigma = \sigma^M \leq |\lambda_j|^M < |\lambda_j| \leq 1 - \omega_\lambda < 1. \tag{3.75}$$

   Noting that $|\lambda_j|^M \neq 0$ again in this case, we substitute (3.75) into (3.73),

   $$\frac{|1 - (1 - \omega_\lambda)|}{\sigma^M(1/\sigma^M + 1)} \leq |\beta_j| \leq \frac{1 + (1 - \omega_\lambda)}{\sigma^M|1/(1 - \omega_\lambda) - 1|}.$$

   Simplifying,

   $$\frac{\omega_\sigma\omega_\lambda}{1 + \omega_\sigma} \leq |\beta_j| \leq \frac{\omega_\sigma(2 - \omega_\lambda)(1 - \omega_\lambda)}{\omega_\lambda}. \tag{3.76}$$

Relaxing the upper and lower limits in (3.69), (3.71), (3.74) and (3.76), we have

$$\omega_\lambda/2 < |\alpha_j| < 2/\omega_\lambda, \tag{3.77}$$

$$\omega_\lambda/(1 + \omega_\sigma) \leq |\alpha_j| \leq (1 + \omega_\sigma)/\omega_\lambda, \tag{3.78}$$

$$\omega_\lambda/(2\,\omega_\sigma) < |\beta_j| < 2\,(1 + |\lambda_j|)/\omega_\lambda, \tag{3.79}$$

$$\omega_\lambda/2 < |\beta_j| < 2\,\omega_\sigma/\omega_\lambda. \tag{3.80}$$

Considering (3.77)–(3.80), each diagonal element $[X]_{(j,j)}, j = 1, \ldots, n$ of X must satisfy

$$\omega_\lambda/(2\,\omega_\sigma) < |[X]_{(j,j)}| < 2\max_j[1 + |\lambda_j|, \omega_\sigma]/\omega_\lambda$$

and hence, since X is diagonal,

$$\mathcal{K}_2(X) = \max_j |[X]_{(j,j)}| / \min_j |[X]_{(j,j)}| < 4\,(\omega_\sigma/\omega_\lambda^2)\max_j[1 + |\lambda_j|, \omega_\sigma]. \tag{3.81}$$

Substituting (3.81) into (3.60) gives (3.51). □

3.3.2 Some Examples Where 1.0-RSCALE Fails

We now present several numerical examples illustrating the potential instability in the prototype algorithm 1.0-RSCALE, the effect of a small shift in σ on problems where 1.0-RSCALE fails, and the sharpness of the bounds derived in Lemmas 7, 8 and 10.

Given the identities derived in Lemmas 4, 5 and 6, it is not difficult to construct well-posed problems that cannot be solved stably with 1.0-RSCALE. For example, if any eigenvalue ν_j of V in (3.11) is a root of the equation

$$(-r)^{M-k^*+1} - 1 = 0, \tag{3.82}$$

then, considering (3.19), (3.21) and (3.23), clearly there will be an exact singularity at the computation of $\hat{V}_{k^*}$, $\tilde{V}_{k,k^*-1}$ and $\tilde{W}_{k^*,i}$, respectively. In fact, potentially there are many such singularities. Any ν_j satisfying (3.82) *always* satisfies

$$(-r)^{c(M-k^*+1)} - 1 = 0,\ c = 2, \ldots, \lfloor M/(M-k^*+1) \rfloor, \tag{3.83}$$

leading to additional singularities in the computation of $\hat{V}_{M-c(M-k^*+1)+1}$, $\tilde{V}_{k,M-c(M-k^*+1)}$ and $\tilde{W}_{M-c(M-k^*+1)+1,i}$ at $c = 2, \ldots, \lfloor M/(M-k^*+1) \rfloor$. Also, when (3.82) is of high degree (i.e. $k^* \ll M$), a ν_j satisfying (3.82) *may* satisfy

$$(-r)^{c(M-k^*+1)} - 1 = 0,\ c < 1,\ c(M-k^*+1) \in \mathcal{N}, \tag{3.84}$$

leading to singularities in the computation of $\hat{V}_{k^\dagger}$, $\tilde{V}_{k,k^\dagger-1}$ and $\tilde{W}_{k^\dagger,i}$, at one or more $k^\dagger > k^*$.

There are $M - k^* + 1$ roots in (3.82), most of which are complex. If one of these roots appears in the spectrum of a *real* matrix V, so must its complex conjugate. Thus, a strategy for constructing a problem that cannot be solved stably with 1.0-RSCALE is to work "backwards": First choose a k^*, then determine the roots of (3.82), and then select an appropriate number of conjugate pairs of these roots as eigenvalues for V. The test problems in Table 3.1 are constructed in this manner. Once eigenvalues $[\nu_1, \ldots, \nu_i, r_{j_1}, \bar{r}_{j_1}, \ldots, r_{j_{(n-i)/2}}, \bar{r}_{j_{(n-i)/2}}]$ have been selected, $V \in \mathcal{R}^{n\times n}$ is formed as

$$V = S\,\mathrm{diag}\{\nu_1, \ldots, \nu_i, R_{j_1}, \ldots, R_{j_{(n-i)/2}}\}S^{-1},$$

where

$$R_{j_l} = \begin{bmatrix} \mathrm{real}\{r_{j_l}\} & -\mathrm{imag}\{r_{j_l}\} \\ \mathrm{imag}\{r_{j_l}\} & \mathrm{real}\{r_{j_l}\} \end{bmatrix},\ l = 1, \ldots, (n-i)/2,$$

and $S \in \mathcal{R}^{n\times n}$ is randomly generated. Note that the rows of S^{-1} are a purely real linear combination of the left eigenvectors of V. $\mathcal{B}_a$ and $\mathcal{B}_b$ in (3.11) are then formed from the rows

Table 3.1: Seven test problems for σ-RSCALE

Selected eigenvalues for V are $[\nu_1, \ldots, \nu_i, r_{j_1}, \bar{r}_{j_1}, \ldots, r_{j_{(n-i)/2}}, \bar{r}_{j_{(n-i)/2}}]$, where r_{j_l} is the j_l-th root of $(-r)^{M-k^*+1} - 1 = 0$, and $\bar{r}_{j_l}$ is the complex conjugate of r_{j_l}.

Problem #	(n, M)	Selected eigenvalues for V	k^*
A	(6, 64)	$[-2.7, -1.7, -1.4, -0.7, -0.4, -0.2]$	–
B	(6, 64)	$[-2.7, -1.7, -0.4, -0.2, r_2, \bar{r}_2]$	50
C	(6, 64)	$[-2.7, -0.2, r_2, \bar{r}_2, r_{17}, \bar{r}_{17}]$	42
D	(8, 32)	$[-2.7, -0.2, r_2, \bar{r}_2, r_{17}, \bar{r}_{17}, r_{27}, \bar{r}_{27}]$	2
E	(4, 128)	$[-2.7, -0.2, r_{107}, \bar{r}_{107}]$	4
F	(6, 64)	$[-2.7, -1.7, -0.4, -0.2, r_{47}, \bar{r}_{47}]$	1
G	(5, 64)	$[-2.7, -1.0, -1.0, -1.0, -0.2]$	–

of S^{-1} corresponding to decreasing ($|\nu_j| < 1$) and increasing or neutral ($|\nu_j| \geq 1$, r_{j_l} and $\bar{r}_{j_l}$) solution modes, respectively. Assuming that $\|V\|_2$ is not too large, an ABD matrix constructed in this manner is always well-conditioned [Asch 88].

The right-hand-side of each test problem is set up in such a way that the resulting linear system represents the trapezoidal finite-difference discretization of (3.10) with

$$q(t) = [I - A] \begin{bmatrix} e^t \\ \vdots \\ e^t \end{bmatrix}$$

and

$$g = \mathcal{B}_a \begin{bmatrix} e^a \\ \vdots \\ e^a \end{bmatrix} + \mathcal{B}_b \begin{bmatrix} e^b \\ \vdots \\ e^b \end{bmatrix}.$$

With this particular choice for $q(t)$ and g, the analytic solution to (3.10) is

$$y(t) = \begin{bmatrix} e^t \\ \vdots \\ e^t \end{bmatrix}. \tag{3.85}$$

Again, the strategy for constructing such a right-hand-side is to work backwards; first choose an appropriate interval of integration $[t_a, t_b]$, and then compute A from V by applying the inverse

of the trapezoidal formula:

$$A = \left(\frac{2M}{b-a}\right)[V+I]\,[V-I]^{-1}.$$

We now present the results of several numerical experiments run under Matlab 5.1 on a 167MHz Sun SPARCstation Ultra 2. The test problems in Table 3.1 are each solved using σ-RSCALE, with various choices for σ, and the results of each experiment are summarized in a figure containing a table and four plots. The table at the top of the figure gives some statistics specific to that experiment; namely ω_σ and ω_λ from Criteria 1 and 2, the theoretical bounds for $\|\tilde{V}_{k,i}\|_2$, $\|\tilde{W}_{k,i}\|_2$, and $\mathcal{K}_2(C)$ derived in Lemmas 7, 8 and 10, respectively, and the analytic and algebraic error in the computed solution $\bar{Y}$. The analytic error is measured as $\|\bar{Y} - Y_{\text{ana}}\|_\infty$, where $y_{\text{ana},i} = y(t_i)$ is the analytic solution (3.85) evaluated at mesh point t_i. The algebraic error is measured as $\|\bar{Y} - Y_{\text{alg}}\|_\infty$, where Y_{alg} is a solution to (3.11) obtained using Matlab's implementation of Gaussian elimination. To help assess the accuracy of the computed solution, h^2 and $\bar{\mathcal{K}}_1(\mathcal{J})$—the 1-norm condition number estimate of the full matrix in (3.11)—are listed beside the analytic and algebraic errors. The four plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right). These plots of sample norms arising in the actual algorithm are included to help gauge the accuracy of the theoretical bounds, and to illustrate exactly where the algorithm fails when singularities occur. Note that the eigenvalue-shift plot is rather trivial in the static variant of the algorithm since σ remains constant across the problem domain. This plot is more enlightening in the dynamic variant of the algorithm presented in §3.3.4.

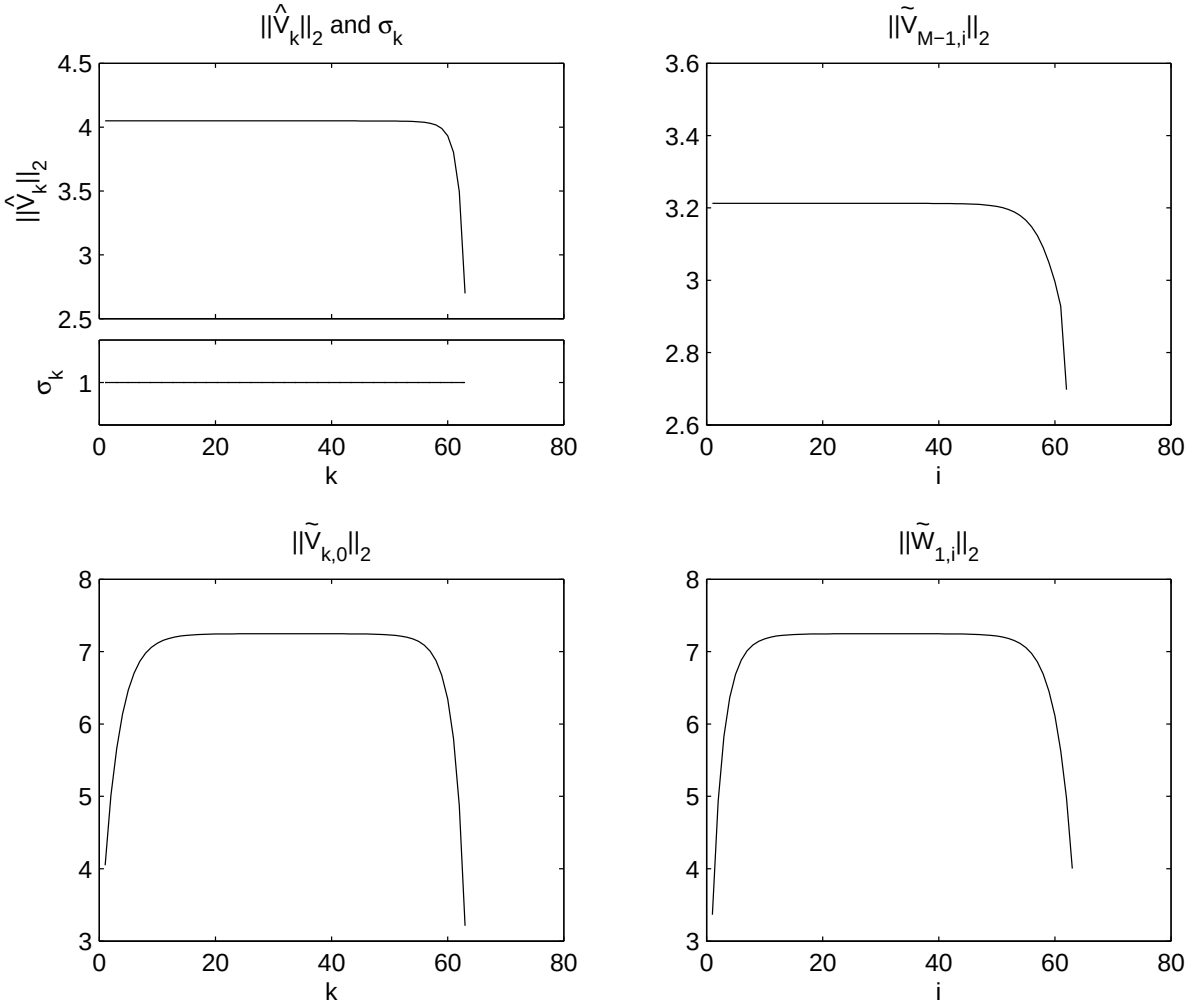
Figure 3.6 summarizes the 1.0-RSCALE solution to Problem A of Table 3.1. With $\omega_\lambda = 0.3$, each eigenvalue ν_j of V is sufficiently greater than 1 in magnitude, so we expect 1.0-RSCALE to be stable on this problem and give an accurate solution. The analytic and algebraic errors listed in the table show this indeed is the case. The plots show that the theoretical bounds on the norms are each about an order of magnitude greater than the actual values; the same is true for the theoretical bound on $\mathcal{K}_2(C)$.

Figure 3.7 summarizes the 1.0-RSCALE solution to Problem B of Table 3.1. This problem is constructed in such a way that the algorithm fails due to a singularity at the computation of rescaled $\hat{V}_{50}$. The plot of $\|\hat{V}_k\|_2$ confirms this singularity, and uncovers three others at the computation of $\hat{V}_{35}$, $\hat{V}_{20}$ and $\hat{V}_5$. These additional singularities arise because eigenvalues ν_5 and ν_6 of V (i.e. r_2 and $\bar{r}_2$) not only satisfy (3.82), but also (3.83) for $c = 2, 3$ and 4. Corresponding singularities in the computation of $\tilde{V}_{M-1,49}$, $\tilde{V}_{M-1,34}$, $\tilde{V}_{M-1,19}$, and $\tilde{V}_{M-1,4}$ are shown in the plot

Figure 3.6: The static 1.0-RSCALE solution to Table 3.1/A.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 0.3	2.6e+02	2.6e+02	1.5e+03	0.00098	0.00029	9.9e+02	3.9e-13

The true value of $\mathcal{K}_2(C)$ is 49. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):



of $\|\tilde{V}_{M-1,i}\|_2$. These last four singularities are precisely the source of instability in the computation of $\tilde{\phi}_{M-1}$ in (3.14). Similar singularities arise in the computation of $\tilde{\phi}_k$, $k = 1, \ldots, M-2$ in (3.14), and $\hat{\phi}_{50}$, $\hat{\phi}_{35}$, $\hat{\phi}_{20}$ and $\hat{\phi}_5$ in (3.12). The effect is a loss of accuracy sufficient to render the computed solution $\bar{Y}$ worthless, even though in this problem the compacted matrix C happens to be well-conditioned, and, as shown in the plot of $\tilde{V}_{k,0}$, expansion (3.16) is stable for all $\tilde{y}_k$.

Figure 3.8 summarizes the 1.02-RSCALE solution to Problem B of Table 3.1. A small shift in σ of $+0.02$ is sufficient to evade the singularities arising in 1.0-RSCALE, and as indicated by the algebraic and analytic errors, the computed solution is now acceptably accurate. Note that since ω_σ does not grow too large, instabilities attributable to large ω_σ do not occur. The danger of over-shifting when evading singularities is addressed in §3.3.4.

The 1.0, 0.98 and 1.02-RSCALE solutions to Problems C-F of Table 3.1 are summarized in Appendix A. In all cases, singularities which arise in 1.0-RSCALE as predicted by the above analysis do not occur in either 0.98 or 1.02-RSCALE. Other details specific to these experiments are given in the introduction to the appendix.

A final comment concerning Table 3.1: Each problem in this table possesses a dichotomy sufficiently "strong" to cause instability in unmodified compactification (i.e. compactification *without* rescaling, or BlkCR). For example, consider the 0.0-RSCALE (BlkCR) solution to Problem B summarized in Figure 3.9. When $\sigma = 0$, $\{\hat{W}_k = I, \hat{V}_k = V, k = 1, \ldots, M\}$, resulting in constant plots for $\|\hat{V}_k\|_2$ and $\|\tilde{W}_{1,i}\|_2$ across the problem domain. The plots for $\|\tilde{V}_{M-1,i}\|_2$ and $\|\tilde{V}_{k,0}\|_2$, on the other hand, clearly show the instability of the unmodified algorithm on this problem.

3.3.3 How Often Does 1.0-RSCALE Fail?

Whether or not 1.0-RSCALE is stable on a given BVP depends both on the system of ODEs and the underlying numerical method used to discretize the continuous problem, making it difficult to predict just how often the algorithm will fail in practice. It is possible, however, to roughly predict the frequency of instability in specific cases. To this end, we now look more closely at ABD systems arising from the trapezoidal finite-difference discretization of (3.10).

When (3.11) is constructed using the trapezoidal scheme, eigenvalues ν_j of V are given by

$$\nu_j = -\left[1 - \frac{h}{2}\alpha_j\right]^{-1}\left[1 + \frac{h}{2}\alpha_j\right], \; j = 1, \ldots, n, \tag{3.86}$$

Figure 3.7: The static 1.0-RSCALE solution to Table 3.1/B.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 2e-15	3.4e+16	3.4e+16	3.4e+31	0.00098	6.2e+13	2.8e+04	6.2e+13

The true value of $\mathcal{K}_2(C)$ is 50. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

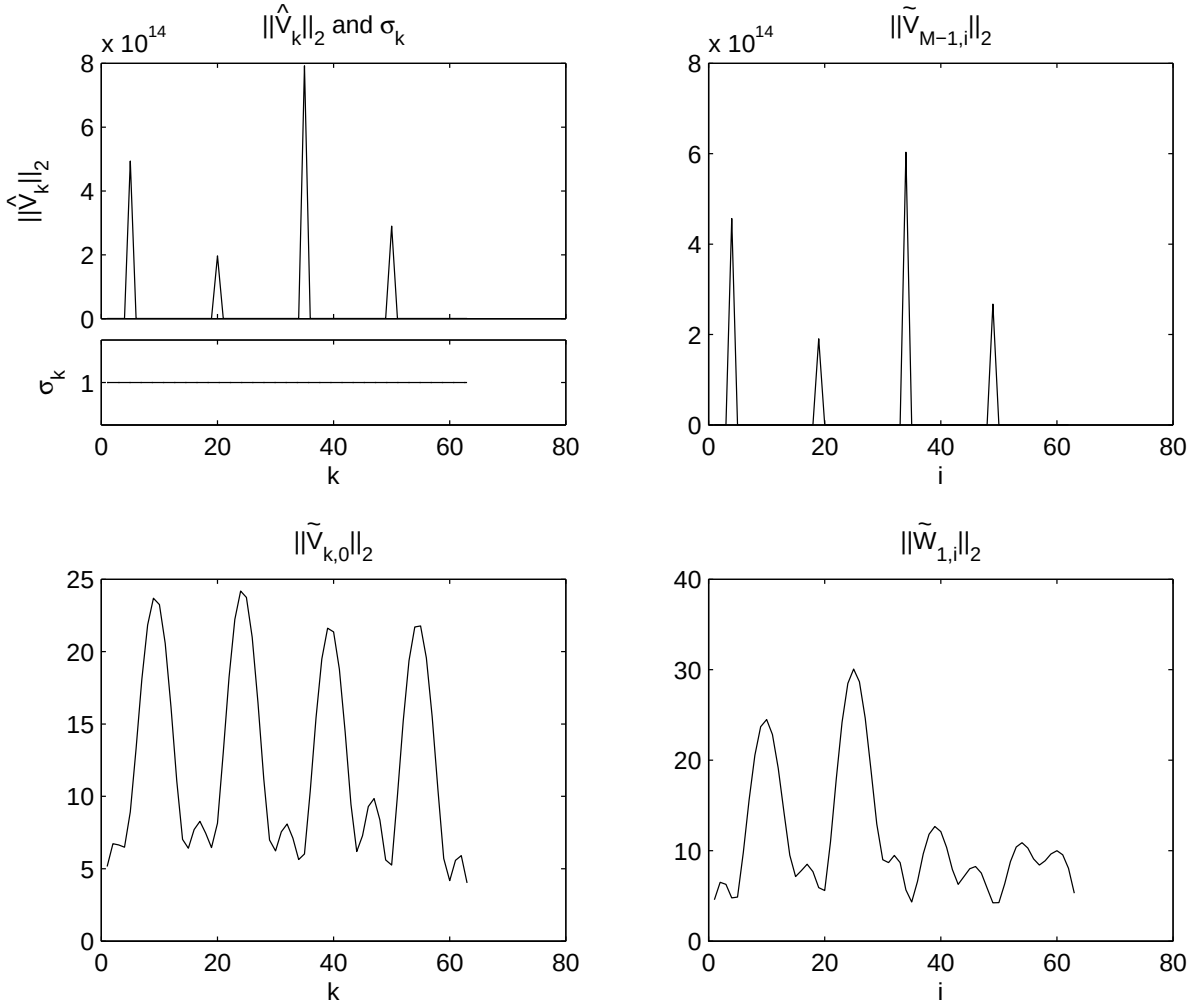


Figure 3.8: The static 1.02-RSCALE solution to Table 3.1/B.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.6, 0.02	3.4e+03	3.4e+03	4.3e+06	0.00098	0.00033	2.8e+04	2.2e-11

The true value of $\mathcal{K}_2(C)$ is 72. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

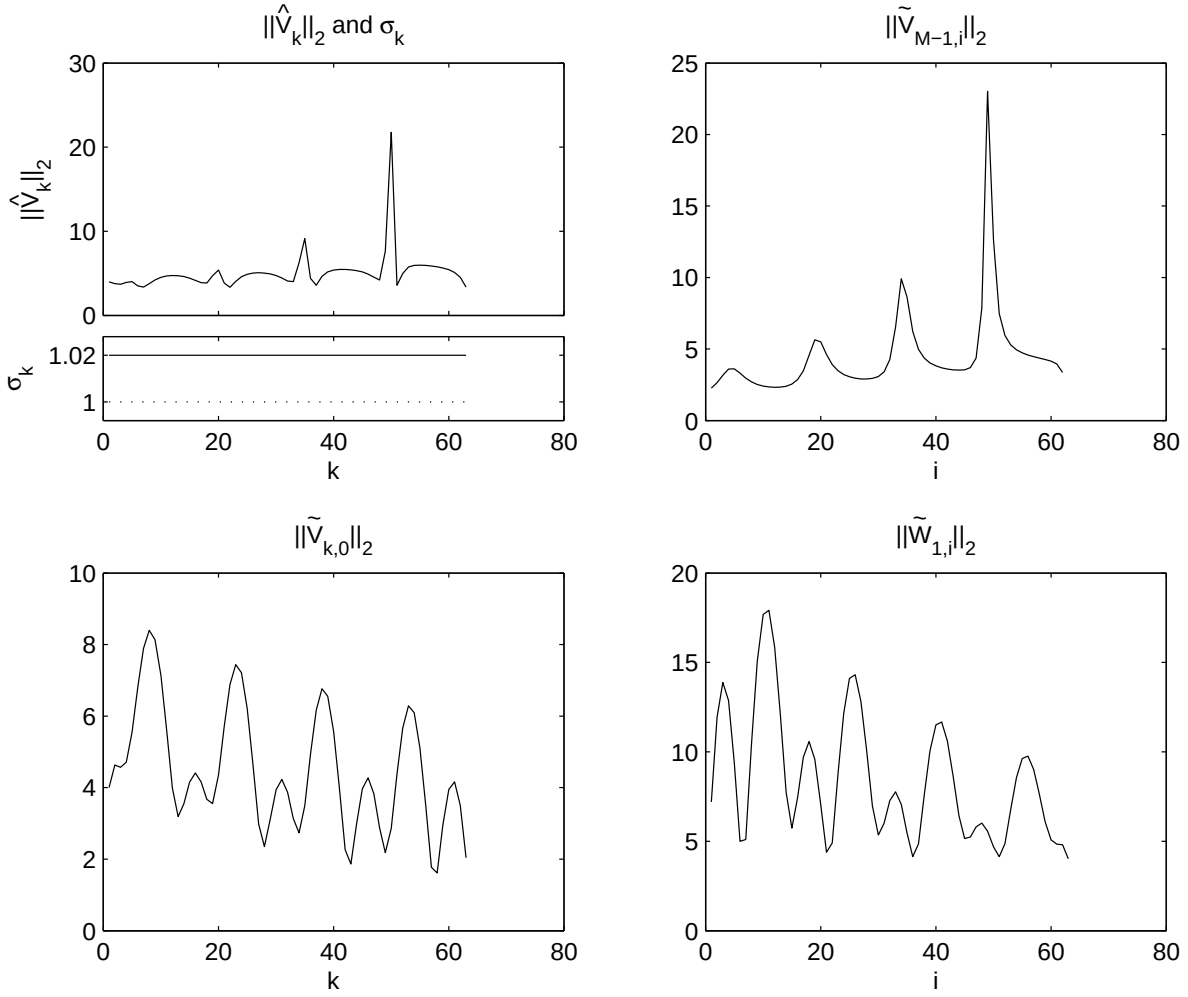
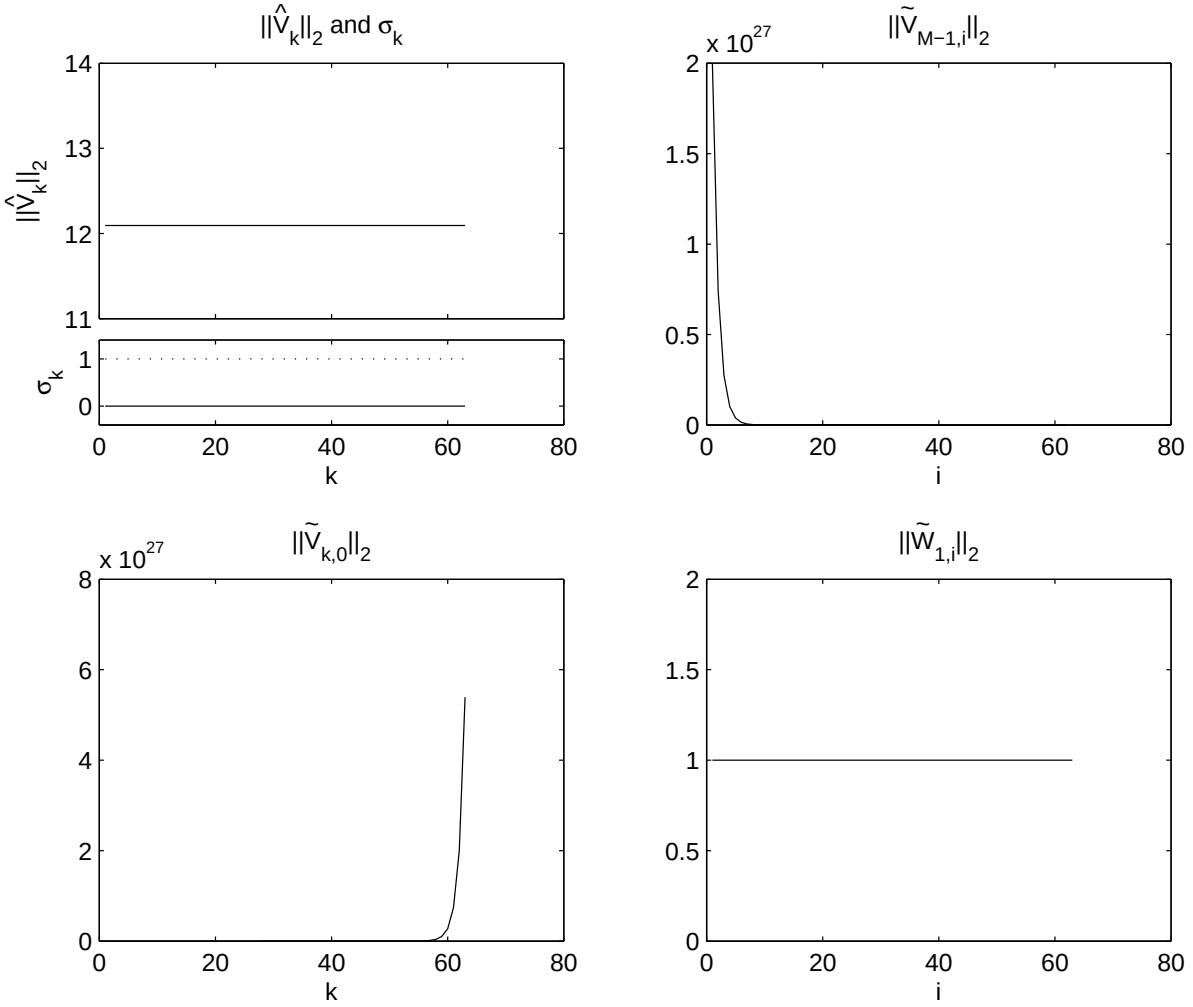


Figure 3.9: The static 0.0-RSCALE (BlkCR) solution to Table 3.1/B.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
$\infty,\ 1$	∞	∞	∞	0.00098	6.9e+11	2.8e+04	6.9e+11

The true value of $\mathcal{K}_2(C)$ is ∞. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):



where α_j is the j-th eigenvalue of A. If $\alpha_j = \alpha_j^{(r)} + i\,\alpha_j^{(i)}$ is complex,

$$\begin{aligned}\nu_j &= -\left[1 - \frac{h}{2}(\alpha_j^{(r)} + i\,\alpha_j^{(i)})\right]^{-1}\left[1 + \frac{h}{2}(\alpha_j^{(r)} + i\,\alpha_j^{(i)})\right] \\ &= -\left[(1 - \frac{h}{2}\alpha_j^{(r)}) - i\,\frac{h}{2}\alpha_j^{(i)}\right]^{-1}\left[(1 + \frac{h}{2}\alpha_j^{(r)}) + i\,\frac{h}{2}\alpha_j^{(i)}\right]\end{aligned}$$

and

$$|\nu_j|^2 = \left[(1 - \frac{h}{2}\alpha_j^{(r)})^2 + (\frac{h}{2}\alpha_j^{(i)})^2\right]^{-1}\left[(1 + \frac{h}{2}\alpha_j^{(r)})^2 + (\frac{h}{2}\alpha_j^{(i)})^2\right].$$

Clearly,

$$|\nu_j| = 1 \Longleftrightarrow \alpha_j^{(r)} = 0 \ \vee \ \alpha_j = 0 \tag{3.87}$$

and

$$|\nu_j| \approxeq 1 \Longleftrightarrow \alpha_j^{(r)} \approxeq 0 \ \vee \ \alpha_j \approxeq 0 \ \vee \ h \approxeq 0. \tag{3.88}$$

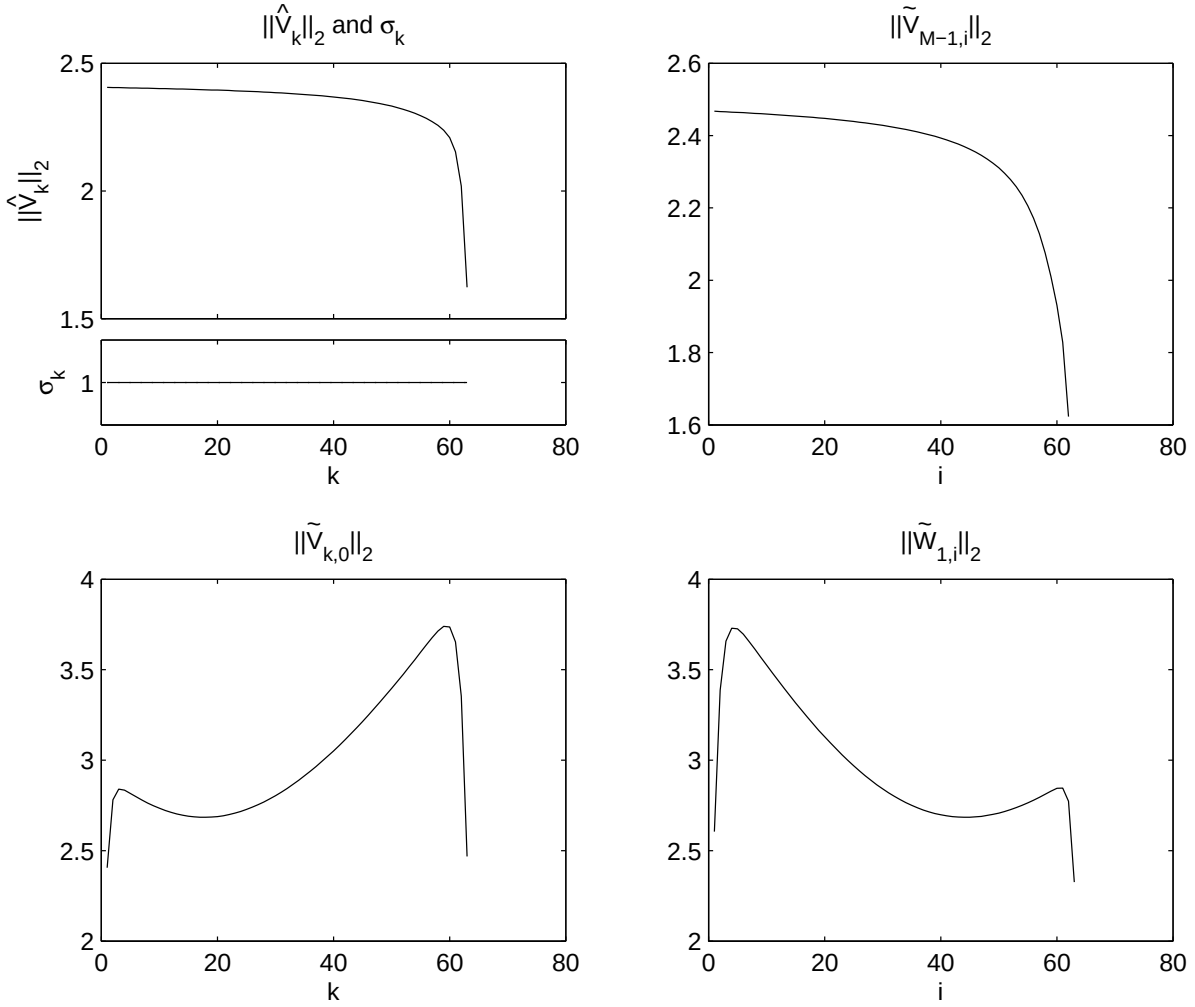
(We omit $h = 0$ in (3.87) since $h > 0$ in every finite discretization. Also, we choose to omit $(h/2)\alpha_j \to \infty$ in (3.88), since this is not likely to occur in any "reasonable" discretization.)

Now referring to Criterion 2 and Lemmas 7, 8 and 10, we may expect instability in σ-RSCALE when $\omega_\lambda = 0$ or $\omega_\lambda \approxeq 0$, which, in 1.0-RSCALE, occurs when $|\nu_j| = 1$ or $|\nu_j| \approxeq 1$. Given (3.87) and (3.88), this occurs in the trapezoidal scheme when any eigenvalue α_j of A has real part exactly or nearly 0, both real and imaginary parts exactly or nearly 0, or when h is nearly 0. This prediction, however, is somewhat pessimistic. There are at least two reasons why the algorithm itself may remain stable even though the theoretical bounds "blow up" due to a small ω_λ. First, an eigenvalue ν_j of V of magnitude 1 is not necessarily a root of (3.82) for some $k^* < M$; i.e. such an eigenvalue will not necessarily cause a singularity in the computation of $\hat{V}_k$, $\tilde{V}_{k,i}$ and $\tilde{W}_{k,i}$ within the problem domain $1 \le k \le M-1,\ 0 \le i \le k-1$. Given that there are infinitely many complex numbers of magnitude 1, the likelihood of one satisfying (3.82) is not great. Second, and perhaps more importantly, the eigenvalue $\nu_j = -1$ or $\nu_j \approxeq -1$, which arises in the trapezoidal scheme when $\alpha_j = 0$, $\alpha_j \approxeq 0$, or h is small, is *not* a source of instability in 1.0-RSCALE. This is an *extraneous* singularity introduced in the derivation of (3.19) in Lemma 4. (The term $[I + \sigma V]^{-1}[I + \sigma V]$ was inserted during the "base" step of the induction to simplify the algebra.) Figure 3.10 summarizes the 1.0-RSCALE solution to Problem G of Table 3.1, in which three of five eigenvalues of V are -1. The theoretical bounds "blow up", but as shown by the analytic and algebraic errors and plots, 1.0-RSCALE is stable and the computed solution is accurate. This is fortunate, since $\nu_j \approxeq -1$ arises in *every* reasonable discretization when h is sufficiently small.

Figure 3.10: The static 1.0-RSCALE solution to Table 3.1/G.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 0	∞	∞	∞	0.00098	0.00027	6e+03	5.5e-13

The true value of $\mathcal{K}_2(C)$ is 8.8e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):



Therefore, more precisely, we may expect instability in 1.0-RSCALE if any eigenvalue α_j of A is exactly or nearly purely imaginary, and the corresponding ν_j satisfies or nearly satisfies (3.82) for some $k^* < M$. With this more accurate prediction, it is not surprising that in the hundreds of random tests on ABD systems arising from the trapezoidal discretization reported in Chapter 4, the prototype algorithm 1.0-RSCALE did not once fail.

3.3.4 Dynamic σ_k-RSCALE

Although the static σ-RSCALE algorithm is simple enough to analyze, its implementation suffers from at least two shortcomings. First, in general it is impossible to determine *a priori* a suitable value for σ when solving an ABD system arising from the discretization of a variable-coefficient BVODE. In an implementation of σ-RSCALE designed to solve such a problem, it is necessary to initially "guess" a value for σ (likely $\sigma = 1$), attempt to rescale with this value, and, if rescaling fails, start over again with a new σ. Although failure is not likely with $\sigma = 1$, this approach has the potential to prove costly. Second, it is not always possible—even in the constant-coefficient case—to choose a σ that guarantees stability. Consider a problem in which both n and M are large, and all or most eigenvalues of V are complex and clustered around the unit-circle. Although it is *always* possible to choose a σ to shift these eigenvalues a "safe" distance from the unit-circle, the required σ could easily result in a prohibitively large $\omega_\sigma = \sigma^M$ (especially if M is large), causing instability in the computation of (3.12) and a poorly conditioned compacted matrix C (Lemma 10). To illustrate the negative effects of a large ω_σ, consider Problem B of Table 3.1. Since neither n nor M is particularly large in this problem, and only 2 eigenvalues of V are clustered on the unit-circle, it is possible to compute an accurate solution with a modest value for σ (Figure 3.8). When the same problem is solved with $\sigma = 1.7$, however, accuracy is lost (Figure 3.11). Comparing the plots in the two Figures, it is evident that $\|\tilde{V}_{k,i}\|_2$ is significantly *damped* by the larger σ. This by itself is a positive effect, in that it could only improve stability in the computation of recurrences (3.14) and (3.16). Unfortunately, however, any improvement is usually far outweighed by the negative effects of a large ω_σ on other stages of the algorithm; namely recurrence (3.12) and the solution of the compacted linear system in step 3(a) of Figure 3.5.

These weaknesses in static σ-RSCALE are addressed in the design of the *dynamic* variant of the algorithm presented in Figure 3.12. In σ_k-RSCALE, the eigenvalue shifts are continually adjusted during rescaling. The frequency and magnitude of the adjustments is determined by bounding the growth of $\|\hat{V}_k\|_1$. Specifically, the following criterion is met in σ_k-RSCALE:

Figure 3.11: The static 1.7-RSCALE solution to Table 3.1/B.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
5.6e+14, 0.32	2.5e+02	2.5e+02	4.2e+32	0.00098	24	2.8e+04	24

The true value of $\mathcal{K}_2(C)$ is 1.9e+15. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

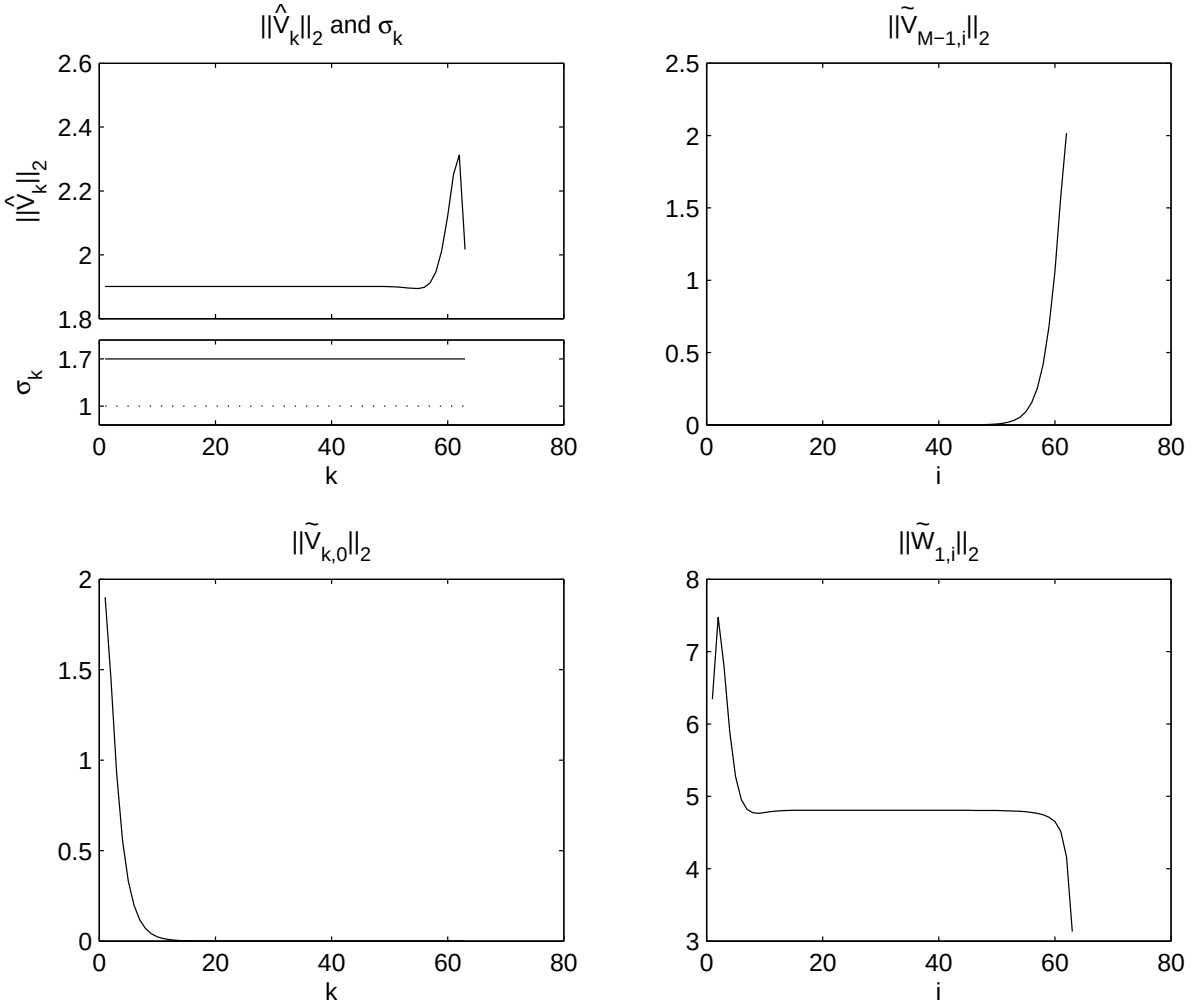


Figure 3.12: Dynamic σ_k-RSCALE applied to the model ABD system (3.11).

0. **Initialization.** $\sigma_k = 1,\ k = 1, 2, \ldots, M-1.$

1. **Rescaling.**

   (a) $\hat{W}_{M-1} = [I - \sigma_{M-1}V],\ \ \hat{V}_{M-1} = \hat{W}_{M-1}^{-1}V.$
   while $\|\hat{V}_{M-1}\|_1 > \tau_{\hat{V}}:\ \sigma_{M-1} = \sigma_{M-1} + \epsilon_\sigma,$
   $\hat{W}_{M-1} = [I - \sigma_{M-1}V],\ \ \hat{V}_{M-1} = \hat{W}_{M-1}^{-1}V$ **end while**.
   $\hat{\phi}_{M-1} = \hat{W}_{M-1}^{-1}\phi_{M-1}.$

   (b) **for** $k = M-2, M-3, \ldots, 1:$
   $\hat{W}_k = [I - \sigma_k V + \sigma_{k+1}\hat{V}_{k+1}],\ \ \hat{V}_k = \hat{W}_k^{-1}V,$
   while $\|\hat{V}_k\|_1 > \tau_{\hat{V}}:\ \sigma_k = \sigma_k + \epsilon_\sigma,$
   $\hat{W}_k = [I - \sigma_k V + \sigma_{k+1}\hat{V}_{k+1}],\ \ \hat{V}_k = \hat{W}_k^{-1}V$ **end while**,
   $\hat{\phi}_k = \hat{W}_k^{-1}(\phi_k + \sigma_{k+1}\hat{\phi}_{k+1}),$
   end for.

   (c) $\hat{\mathcal{B}}_a = \mathcal{B}_a(I + \sigma_1\hat{V}_1),\ \ \hat{g} = g + \sigma_1\mathcal{B}_a\hat{\phi}_1.$

2. **Compactification.**

   (a) $\tilde{V}_1 = \hat{V}_1,\ \ \tilde{\phi}_1 = \hat{\phi}_1.$

   (b) $\left.\begin{array}{l} \tilde{V}_k = -\hat{V}_k\tilde{V}_{k-1} \\ \tilde{\phi}_k = \hat{\phi}_k - \hat{V}_k\tilde{\phi}_{k-1} \end{array}\right\}\ k = 2, 3, \ldots, M-1.$

   (c) $\tilde{V}_M = -V\tilde{V}_{M-1},\ \ \tilde{\phi}_M = \phi_M - V\tilde{\phi}_{M-1}.$

3. **Computation of transformed unknowns.**

   (a) $\tilde{y}_0 = [\hat{\mathcal{B}}_a - \mathcal{B}_b\tilde{V}_M]^{-1}(\hat{g} - \mathcal{B}_b\tilde{\phi}_M).$

   (b) $\tilde{y}_k = \tilde{\phi}_k - \tilde{V}_k\tilde{y}_0,\ k = 1, 2, \ldots, M.$

4. **Recovery of original unknowns.**

   (a) $y_{k-1} = \tilde{y}_{k-1} - \sigma_k\tilde{y}_k,\ k = 1, 2, \ldots, M-1.$

   (b) $y_{M-1} = \tilde{y}_{M-1},\ \ y_M = \tilde{y}_M.$

Criterion 6 $\exists\, \tau_{\hat{V}} > 0$ *such that* $\|\hat{V}_k\|_1 \leq \tau_{\hat{V}}$ *for all* $k, 1 \leq k \leq M-1$.

(The 1-norm is used in lieu of the 2-norm to reduce overhead.) Initially, each eigenvalue shift σ_k is set to 1. In order to satisfy Criterion 6 in steps 1(a) and 1(b) of Figure 3.12, however, it may be necessary to adjust one or more σ_k. The adjustment could be determined exactly given the eigenvalues of V and $\hat{V}_{k+1}$, but this is unnecessarily costly. Instead, σ_k is simply incremented by ϵ_σ until Criterion 6 is met. Numerical experiments show that, if ϵ_σ is large enough, usually only one increment is necessary. Note that the "while" loops in steps 1(a) and 1(b) will always terminate if $\epsilon_\sigma > 0$ and $\tau_{\hat{V}} > 0$, since $\hat{W}_k$ has only a finite number of eigenvalues. Whether or not there are optimal values for these parameters, however, is currently an open question. Preliminary analysis indicates that $\tau_{\hat{V}} = \|V_k\|_1$, $\epsilon_\sigma = \|V_k\|_1/n$ are effective choices.

We believe that Criterion 6 is analogous to Criterion 2 in σ-RSCALE, in that singularities will not arise during rescaling if $\|\hat{V}_k\|_1 \approxeq \|V_k\|_1$, $k = 1, \ldots, M-1$. A further analysis of exactly how Criterion 6 affects the stability of σ_k-RSCALE is left for future work.

Figure 3.13 summarizes the σ_k-RSCALE solution to Problem B of Table 3.1. Note that Criterion 6 ($\tau_{\hat{V}} = 10$) is satisfied with only 3 adjusted eigenvalue shifts. Many other numerical experiments show this is typical—when singularities occur in static 1.0-RSCALE, usually only a few adjusted shifts are required to avoid them. The σ_k-RSCALE solutions to Problems C-F of Table 3.1 are summarized in Appendix A.

The expansions for $\tilde{\phi}_k$ in step 2 and $\tilde{y}_k$ in step 3 of σ_k-RSCALE are identical to (3.14) and (3.16) of σ-RSCALE, respectively. The expansion for $\hat{\phi}_k$ in step 1 is a generalization of (3.12):

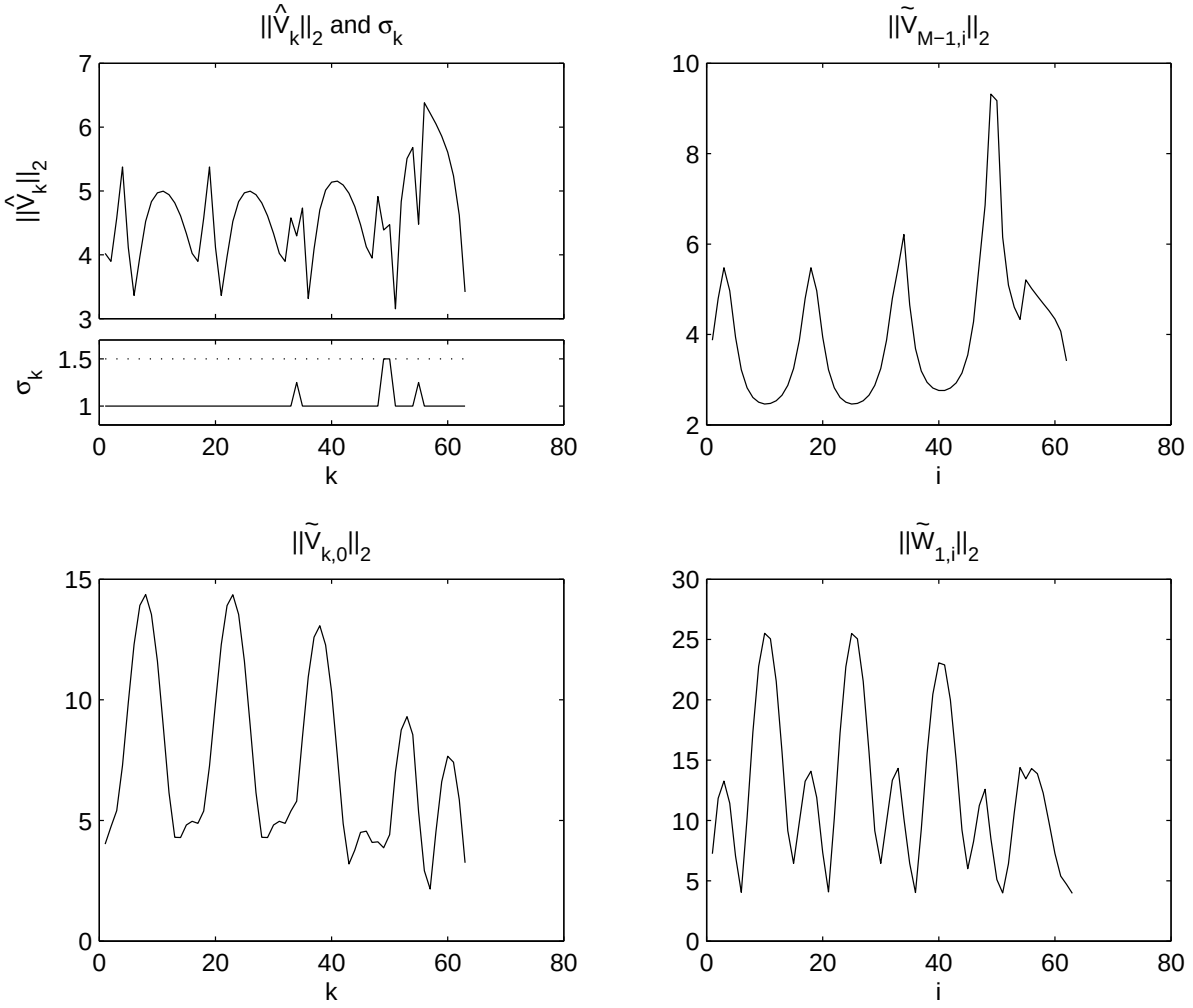
$$\hat{\phi}_k = \sum_{i=k}^{M-1} \left[\prod_{j=k+1}^{i} \sigma_j \right] \left[\prod_{j=k}^{i} \hat{W}_j^{-1} \right] \phi_i \tag{3.89}$$

In σ_k-RSCALE, $\omega_\sigma = \prod_{k=1}^{M-1} \sigma_k$ will not grow too large when $\sigma_k = 1$ for most k, and hence the instability in (3.89) attributable to large ω_σ is avoided. In addition, identities corresponding to Lemmas 4, 5, and 6 exist for σ_k-RSCALE, albeit more complex. We can therefore derive bounds for $\|\tilde{V}_{k,i}\|_2$, $\|\tilde{W}_{k,i}\|_2$ and $\mathcal{K}_2(C)$ similar to those in Lemmas 7, 8, and 10; this, however, is left for future work.

Figure 3.13: The σ_k-RSCALE solution ($\tau_{\hat{V}} = 10.0,\ \epsilon_\sigma = 0.25$) to Table 3.1/B.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.5, ?	?	?	?	0.00098	0.00033	2.8e+04	1.6e-11

The true value of $\mathcal{K}_2(C)$ is 51. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.89) (bottom right):



Chapter 4

Performance of the Algorithms

In this chapter we discuss and assess the performance of the three parallel ABD system solvers *SLF*-QR, *SLF*-LU, and RSCALE, looking not only at their relative performance but also how they compare to some state-of-the-art sequential solvers. Each of the parallel solvers has been implemented in FORTRAN 77, with extensive use of level-3 BLAS. The code is included in Appendix E. Both the accuracy and speed of the implementations are assessed.

In §4.1 we derive operation counts for the computationally-intensive stages of each algorithm: factorization and reduction. Operation counts for other ABD system solvers, such as COLROW [Diaz 83], are cited from the literature. Ratios of the high-order coefficients of these operation counts are used to verify timing measurements reported in numerical experiments throughout the chapter. In §4.2 we assess the performance of the single-partition sequential variant of each algorithm by testing the codes on a single-processor machine. In §4.3 we assess the parallel variants by testing on a shared-memory multi-processor machine. We conclude in §4.4 by assessing the relative performance of the algorithms when the codes are incorporated in MirkDC [Enri 96], a software package for solving nonlinear boundary value ordinary differential equations.

We note that the tests in §4.2 were run several years after the tests run in §4.3 and §4.4. Computer architecture has of course evolved over the years, and processing speed is now considerably faster. The problems in §4.2 have been scaled appropriately so that absolute execution times are similar to those in earlier tests. In other words, we have solved more computationally-intensive problems in recent tests. Detailed architecture specifications are given at the beginning of each section.

4.1 Operation Counts

In Chapter 2, we propose different variants of each of the ABD system solvers. Some variants make better use of idle processors in order to more fully exploit parallelism (e.g., Figure 2.2 in §2.2.3, Figure 2.3 in §2.3.3); some variants may exhibit better stability properties (e.g., (2.16)-(2.19) in §2.2.4). In most cases, the basic computations performed during a block-step of a given solver are the same for all variants of that solver. In this section, we analyze the complexity of *SLF*-QR, *SLF*-LU and RSCALE by deriving operation counts for the factorization and reduction components of their respective block-steps. Ratios of the high-order coefficients of these operation counts are used to verify some of the timing measurements collected when assessing the performance of the solvers in §4.2, §4.3 and §4.4. As the forward and back-solve components of a block-step do not contribute to the high-order coefficients, we do not include these components in our analysis.

We count floating-point operations (flops) in our analysis throughout this section. As is commonly done, we define a flop to be a multiplication-addition pair.

4.1.1 *SLF*-QR

As discussed in §2.2.1, the computations performed during a block-step of *SLF*-QR include a QR-factorization

$$\begin{bmatrix} T_i \\ S_{i+1} \end{bmatrix} = Q \begin{bmatrix} R_i \\ 0 \end{bmatrix}$$

where $Q \in \mathcal{R}^{2n \times 2n}$ is orthogonal and $R_i \in \mathcal{R}^{n \times n}$ is upper-triangular, followed by a reduction applied across the i-th slice of the ABD matrix:

$$\begin{bmatrix} \cdots & S_i^{(k)} & R_i^{(k)} & T_i^{(k)} & \cdots \\ \cdots & S_{i+1}^{(k)} & & T_{i+1}^{(k)} & \cdots \end{bmatrix} \longleftarrow Q^T \times \begin{bmatrix} \cdots & S_i & T_i & & \cdots \\ \cdots & & S_{i+1} & T_{i+1} & \cdots \end{bmatrix}$$

The above is conceptual only. In practice—and, in particular, in the code included in Appendix E—Q is never explicitly computed. Instead, Householder reflections are applied directly to each of the $3n$ columns of

$$\begin{bmatrix} \cdots & S_i & T_i & & \cdots \\ \cdots & & S_{i+1} & T_{i+1} & \cdots \end{bmatrix}$$

in order to compute the triangularization and reduction. We now analyze the cost of applying these reflections.

First, we triangularize $[T_i^T S_{i+1}^T]^T$ using n reflections, computed and applied as follows:

Stage 1: Let $\underline{c}_1 \in \mathcal{R}^{2n}$ represent the first column of $[T_i^T S_{i+1}^T]^T$, and let $\underline{v} = \underline{c}_1 - (\sqrt{\underline{c}_1^T \underline{c}_1})\ \underline{e}_1$ with $\underline{e}_1$ the first column of the $2n \times 2n$ identity matrix. The Householder reflection $Q_1 = I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v}) \in \mathcal{R}^{2n \times 2n}$ annihilates the first column of $[T_i^T S_{i+1}^T]^T$ below its first element:

$$\begin{aligned} Q_1\underline{c}_1 &= [I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v})]\underline{c}_1 \\ &= \underline{c}_1 - [2(\underline{v}^T\underline{c}_1)/(\underline{v}^T\underline{v})]\underline{v} \\ &= \|\underline{c}_1\|_2\underline{e}_1. \end{aligned}$$

Q_1 is not explicitly formed. Only $\underline{v}$ is computed and stored at a cost of $2n$ flops for $\underline{c}_1^T\underline{c}_1$. In addition, $\underline{v}^T\underline{v}$ is computed at negligible extra cost and stored for future use.

The remaining $n-1$ columns of $[T_i^T S_{i+1}^T]^T$ are also transformed with Q_1:

$$\begin{aligned} Q_1\underline{c}_i &= [I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v})]\underline{c}_i \\ &= \underline{c}_i - [2(\underline{v}^T\underline{c}_i)/(\underline{v}^T\underline{v})]\underline{v} \end{aligned}$$

for $i = 2, \ldots, n$. The cost is $2\,(2n)$ flops per column—$2n$ flops for $\underline{v}^T\underline{c}_i$, and $2n$ flops for the vector subtraction and scalar-vector multiplication. $\underline{v}^T\underline{v}$ is not recomputed.

The total cost for stage 1 is therefore $2n$ flops for computing $\underline{v}$, and $2\,(2n)\,(n-1)$ flops for applying the reflection to columns $2, \ldots, n$ of $[T_i^T S_{i+1}^T]^T$.

Stage 2: The first column of $[T_i^T S_{i+1}^T]^T$ has been annihilated. We now repeat the steps in stage 1, but this time on the $(2n-1) \times (n-1)$ submatrix starting at row 2, column 2. Let $\underline{c}_2 \in \mathcal{R}^{2n-1}$ represent the first column of this submatrix, and let $\underline{v} = \underline{c}_2 - (\sqrt{\underline{c}_2^T\underline{c}_2})\ \underline{e}_1$ with $\underline{e}_1$ the first column of the $(2n-1) \times (2n-1)$ identity matrix. The reflection $Q_2 = I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v}) \in \mathcal{R}^{2n-1 \times 2n-1}$ annihilates the first column of the submatrix:

$$\begin{aligned} Q_2\underline{c}_2 &= [I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v})]\underline{c}_2 \\ &= \underline{c}_2 - [2(\underline{v}^T\underline{c}_2)/(\underline{v}^T\underline{v})]\underline{v} \\ &= \|\underline{c}_2\|_2\underline{e}_1. \end{aligned}$$

Again, Q_2 is not explicitly formed—$\underline{v}$ is computed at a cost of $(2n-1)$ flops for $\underline{c}_2^T\underline{c}_2$, and $\underline{v}^T\underline{v}$ is computed at negligible extra cost and stored for future use.

The remaining $n-2$ columns of the submatrix are also transformed with Q_2:

$$\begin{aligned} Q_2\underline{c}_i &= [I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v})]\underline{c}_i \\ &= \underline{c}_i - [2(\underline{v}^T\underline{c}_i)/(\underline{v}^T\underline{v})]\underline{v} \end{aligned}$$

for $i = 3, \ldots, n$. The cost is $2\,(2n-1)$ flops per column—$(2n-1)$ flops for $\underline{v}^T \underline{c}_i$, and $(2n-1)$ flops for the vector subtraction and scalar-vector multiplication.

The total cost for stage 2 is $(2n-1)$ flops for computing $\underline{v}$ and $2\,(2n-1)\,(n-2)$ flops for applying the reflection to columns $3, \ldots, n$ of the submatrix.

Stage j: For $j = 3, \ldots, n-1$, we process the $(2n-j+1) \times (n-j+1)$ submatrix starting at row j column j. The Householder reflection is computed at a cost of $(2n-j+1)$ flops, and the last $(n-j)$ columns of the submatrix are transformed at a cost of $2\,(2n-j+1)$ flops per column. The total cost for stage j is $(2n-j+1) + 2\,(2n-j+1)\,(n-j)$ flops.

Stage n: The n-th column, starting at row n, is annihilated below its first element. The Householder reflection is computed at a cost of $(n+1)$ flops. As this is the last column to be processed, no other columns are transformed with this reflection.

Thus, the total operation count for computing the reflections is

$$\begin{aligned} 2n + (2n-1) + \cdots + (2n-j+1) + \cdots + (n+1) &= \sum_{i=1}^{2n} i - \sum_{i=1}^{n} i \\ &= (2n)(2n+1)/2 - n(n+1)/2 \\ &= (3/2)n^2 + (1/2)n \ \text{ flops} \end{aligned}$$

and the total operation count for applying the reflections is

$$2(2n)(n-1) + 2(2n-1)(n-2) + \cdots + 2(2n-j+1)(n-j) + \cdots + 2(n+2)(1)$$

$$\begin{aligned} &= 2[(n-1)^2 + (n+1)(n-1)] + 2[(n-2)^2 + (n+1)(n-2)] \\ &\quad + \cdots + 2[(n-j)^2 + (n+1)(n-j)] + \cdots + 2[(1)^2 + (n+1)(1)] \\ &= 2\left[\sum_{i=1}^{n-1} i^2 + (n+1)\sum_{i=1}^{n-1} i\right] \\ &= 2[(n-1)(n)(2n-1)/6 + (n+1)(n-1)(n)/2] \\ &= (5/3)n^3 - n^2 - (2/3)n \ \text{ flops} \end{aligned}$$

for a grand total of

$$(5/3)n^3 + (1/2)n^2 - (1/6)n \ \text{ flops.}$$

Once the triangularization is complete, we use the n reflections to transform

$$\begin{bmatrix} S_i \\ \underline{0} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \underline{0} \\ T_{i+1} \end{bmatrix}. \tag{4.1}$$

Stage 1: Application of the stage 1 reflector across the columns $\underline{c}_i$ of (4.1):

$$\begin{aligned} Q_1\underline{c}_i &= [I - 2(\underline{v}\,\underline{v}^T)/(\underline{v}^T\underline{v})]\underline{c}_i \\ &= \underline{c}_i - [2(\underline{v}^T\underline{c}_i)/(\underline{v}^T\underline{v})]\underline{v} \end{aligned}$$

requires $3n$ flops per column, for a total of $(3n)\,(2n)$ flops. The computation of $\underline{v}^T\underline{c}_i$ requires only n flops instead of $2n$ flops because of the sparsity inherent in the $\underline{0}$ blocks. This savings is not realized in subsequent stages, however, as the structural sparsity is destroyed by the stage 1 reflector—the $\underline{0}$ blocks fill-in immediately. (When structural sparsity is preserved, operation counts are reduced. This is an important factor in the analysis of *SLF*-LU in §4.1.2.)

Stage j: For $j = 2, \ldots, n$, application of the stage j reflector across (4.1) requires $2\,(2n-j+1)$ flops per column, for a total of $2\,(2n-j+1)\,(2n)$ flops.

Thus, the total operation count for applying the reflections to transform (4.1) is

$$(3n)(2n) + 2(2n-1)(2n) + \cdots + 2(2n-j+1)(2n) + \cdots + 2(n+1)(2n)$$

$$\begin{aligned} &= 2(2n)(2n) - n(2n) + 2(2n-1)(2n) + \cdots + 2(2n-j+1)(2n) + \cdots + 2(n+1)(2n) \\ &= 4n\left[\sum_{i=1}^{2n} i - \sum_{i=1}^{n} i\right] - 2n^2 \\ &= 4n\,[(2n)(2n+1)/2 - n(n+1)/2] - 2n^2 \\ &= 6n^3 \text{ flops} \end{aligned}$$

Finally, the total cost of all computations performed during a block-step of *SLF*-QR, including both triangularization and reduction, is $(23/3)n^3 + (1/2)n^2 - (1/6)n$ flops.

4.1.2 *SLF*-LU

The *SLF*-LU block-step is similar to that of *SLF*-QR, except that instead of a QR-factorization we use an LU-factorization

$$\begin{bmatrix} T_i \\ S_{i+1} \end{bmatrix} = L \begin{bmatrix} U_i \\ 0 \end{bmatrix}$$

where $L^{-1} = \tilde{L}_n P_n \cdots \tilde{L}_2 P_2 \tilde{L}_1 P_1 \in \mathcal{R}^{2n\times 2n}$, $\tilde{L}_j$ is an elementary Gauss transformation, P_j is a permutation matrix, and $U_i \in \mathcal{R}^{n\times n}$ is upper-triangular. Conceptually, once L^{-1} is obtained,

it is used to reduce the i-th slice of the ABD matrix:

$$\begin{bmatrix} \cdots & \tilde{S}_i^{(k)} & U_i^{(k)} & \tilde{T}_i^{(k)} & \cdots \\ \cdots & \tilde{S}_{i+1}^{(k)} & & \tilde{T}_{i+1}^{(k)} & \cdots \end{bmatrix} \longleftarrow L^{-1} \times \begin{bmatrix} \cdots & S_i & T_i & & \cdots \\ \cdots & & S_{i+1} & T_{i+1} & \cdots \end{bmatrix}$$

SLF-LU is obviously structurally equivalent to *SLF*-QR. We have accented the transformed blocks ($\tilde{S}_i^{(k)}$ versus $S_i^{(k)}$, etc.) to emphasize that the underlying transformation is *not* equivalent.

As with the orthogonal matrix Q in *SLF*-QR, the Gaussian matrix L in *SLF*-LU is never explicitly computed. Instead, the individual permutations (row interchanges) and Gauss transformations (row combinations) are applied directly to each of the $3n$ columns of

$$\begin{bmatrix} \cdots & S_i & T_i & & \cdots \\ \cdots & & S_{i+1} & T_{i+1} & \cdots \end{bmatrix}$$

in order to compute the triangularization and reduction. We now analyze the cost of applying these transformations.

First, we triangularize $[T_i^T S_{i+1}^T]^T$ using n permutations (possibly) and n Gauss transformations, computed and applied as follows:

Stage 1: Before eliminating, we interchange rows if necessary to ensure that the element in row 1, column 1, is the largest in magnitude of all elements in column 1. We then eliminate all elements in the first column below the first element by subtracting multiples of row 1 from each of the other $(2n-1)$ rows. The cost is $(2n-1)$ divisions to form the multipliers and $(2n-1)(n-1)$ flops to process all elements below row 1 and to the right of column 1.

Stage 2: The first column of $[T_i^T S_{i+1}^T]^T$ has been eliminated. We now repeat the steps in stage 1, but this time on the $(2n-1) \times (n-1)$ submatrix starting at row 2, column 2. If necessary, we interchange rows to ensure that the element in row 2, column 2, is the largest in magnitude of all elements in column 2. Elimination requires $(2n-2)$ divisions to form the multipliers and $(2n-2)(n-2)$ flops to process all elements below row 2 and to the right of column 2.

Stage j: For $j = 3, \ldots, n-1$, we process the $(2n-j+1) \times (n-j+1)$ submatrix starting at row j column j. If necessary, we interchange rows to ensure that the element in row j, column j, is the largest in magnitude of all elements in column j. Elimination requires $(2n-j)$ divisions to form the multipliers and $(2n-j)(n-j)$ flops to process all elements below row j and to the right of column j.

Stage n: If necessary, we interchange rows to ensure that the element in row n, column n, is the largest in magnitude of all elements in column n. Elimination of the n-th column, starting at row $(n+1)$, requires n divisions to form the multipliers. As this is the last column to be processed, no other computations are necessary.

Thus, the total operation count for forming the multipliers is

$$\begin{aligned}(2n-1)+(2n-2)+\cdots+(2n-j)+\cdots+n &= \sum_{i=1}^{2n-1} i - \sum_{i=1}^{n-1} i \\ &= (2n-1)(2n)/2-(n-1)(n)/2 \\ &= (3/2)n^2-(1/2)n \text{ divisions}\end{aligned}$$

and the total operation count for applying the Gauss transformations (i.e., subtracting multiples of one row from another) is

$$(2n-1)(n-1)+(2n-2)(n-2)+\cdots+(2n-j)(n-j)+\cdots+(n+1)(1)$$

$$\begin{aligned}&= (2n-1)^2-n(2n-1)+(2n-2)^2-n(2n-2) \\ &\quad +\cdots+(2n-j)^2-n(2n-j)+\cdots+(n+1)^2-n(n+1) \\ &= \sum_{i=1}^{2n-1} i^2-\sum_{i=1}^{n} i^2-n\left[\sum_{i=1}^{2n-1} i-\sum_{i=1}^{n} i\right] \\ &= (2n-1)(2n)(4n-1)/6-(n)(n+1)(2n+1)/6-n\left[(2n-1)(2n)/2-n(n+1)/2\right] \\ &= (5/6)n^3-n^2+(1/6)n \text{ flops}\end{aligned}$$

for a grand total of

$$(5/6)n^3-n^2+(1/6)n \text{ flops and } (3/2)n^2-(1/2)n \text{ divisions.}$$

Once the triangularization is complete, we use the n permutations and n Gauss transformations to transform

$$\begin{bmatrix} S_i \\ \underline{0} \end{bmatrix} \text{ and } \begin{bmatrix} \underline{0} \\ T_{i+1} \end{bmatrix}. \tag{4.2}$$

We must analyze this step carefully, as the row interchanges used during the LU-factorization could have a significant impact on the efficiency of the transformation. We look at the transformation on $[S_i^T \underline{0}^T]^T$ first, considering a best and worst case scenario.

Best Case: At each stage of the factorization, a row below row n is pivoted up, and never the same row twice. This results in a null row pivoted up in $[S_i^T \underline{0}^T]^T$ at each stage of its

transformation, and the associated Gauss transformation at each stage, when applied to $[S_i^T \underline{0}^T]^T$, would require no flops. (In level-3 BLAS, subtracting a multiple of zero does not require work.) The total flops required is therefore 0.

Worst Case: At the first stage of the factorization, no pivoting occurs or a row above row n is pivoted up. The associated Gauss transformation at the first stage, when applied to $[S_i^T \underline{0}^T]^T$, would completely fill in its lower $n \times n$ block and the fill-in would persist throughout the remaining stages of the transformation. (There is no elimination here; the Gauss transformations are designed to eliminate in $[T_i^T S_{i+1}^T]^T$ only.) When there is complete fill-in, the total operation count for transforming $[S_i^T \underline{0}^T]^T$ is

$$(2n-1)(n) + (2n-2)(n) + \cdots + (2n-j)(n) + \cdots + (n+1)(n) + (n)(n)$$

$$\begin{aligned} &= n\left[\sum_{i=1}^{2n-1} i - \sum_{i=1}^{n-1} i\right] \\ &= n\left[(2n-1)(2n)/2 - (n-1)(n)/2\right] \\ &= (3/2)n^3 - (1/2)n^2 \text{ flops} \end{aligned}$$

Next, we look at the transformation on $[\underline{0}^T T_{i+1}^T]^T$ in (4.2), again considering a best and worst case scenario.

Best Case: At each stage of the factorization, there is either no pivoting or a row above row n is pivoted up. This results in a null pivot row in $[\underline{0}^T T_{i+1}^T]^T$ at each stage of its transformation, and the associated Gauss transformation at each stage, when applied to $[\underline{0}^T T_{i+1}^T]^T$, would require no flops. The total flops required is therefore 0.

Worst Case: At the first stage of the factorization, a row below row n is pivoted up. This results in a dense row pivoted up in $[\underline{0}^T T_{i+1}^T]^T$. The associated Gauss transformation at the first stage, when applied to $[\underline{0}^T T_{i+1}^T]^T$, would completely fill in its upper $n \times n$ block and the fill-in would persist throughout the remaining stages of the transformation. The total operation count for transforming $[\underline{0}^T T_{i+1}^T]^T$ is

$$(2n-1)(n) + (2n-2)(n) + \cdots + (2n-j)(n) + \cdots + (n+1)(n) + (n)(n)$$

$$\begin{aligned} &= n\left[\sum_{i=1}^{2n-1} i - \sum_{i=1}^{n-1} i\right] \\ &= n\left[(2n-1)(2n)/2 - (n-1)(n)/2\right] \\ &= (3/2)n^3 - (1/2)n^2 \text{ flops} \end{aligned}$$

Note that we cannot have a best case scenario for both $[S_i^T \underline{0}^T]^T$ and $[\underline{0}^T T_{i+1}^T]^T$, as

$$\begin{aligned} &\text{best case for } [S_i^T \underline{0}^T]^T \implies \text{worst case for } [\underline{0}^T T_{i+1}^T]^T \\ &\text{best case for } [\underline{0}^T T_{i+1}^T]^T \implies \text{worst case for } [S_i^T \underline{0}^T]^T \end{aligned}$$

We could have a "near" worst case scenario for both, though. For example, at the first stage a dense row is pivoted up in $[\underline{0}^T T_{i+1}^T]^T$ and at the second stage no pivoting occurs. This would result in both $[S_i^T \underline{0}^T]^T$ and $[\underline{0}^T T_{i+1}^T]^T$ being dense from the second stage onward, except for the first row in the former. The transformation costs are then

$$\begin{aligned} [\underline{0}^T T_{i+1}^T]^T\text{:} &\quad (3/2)n^3 - (1/2)n^2 \text{ flops} \\ [S_i^T \underline{0}^T]^T\text{:} &\quad (3/2)n^3 - (1/2)n^2 - (2n-1)n \text{ flops} \end{aligned}$$

In summary, the total cost of all computations performed during a block-step of *SLF*-LU, including both triangularization and reduction, is

$$\mathcal{K}\, n^3 + \mathcal{O}(n^2) \text{ flops},$$

where $\mathcal{K} \in [7/3, 23/6]$, and its exact value depends on the pivoting strategy employed during the LU-factorization.

An interesting question now arises: In a "typical" block-step of *SLF*-LU, does $\mathcal{K}$ tend toward its lower or upper bound? Considering the analysis in §3.2.1, we see immediately that $\mathcal{K}$ achieves its lower bound only in problems where *SLF*-LU is potentially unstable; i.e., when there is no pivoting, no cross-block pivoting, or maximum pivoting causing the block-rows to be flipped. In the experiments in §4.2, §4.3 and §4.4, we see that $\mathcal{K}$ tends toward its upper bound in most problems where *SLF*-LU is stable.

4.1.3 RSCALE

In §2.3.2 and §2.3.3, we introduced RSCALE by showing how its transformations are applied to the right-block-identity form of the ABD system (1.9). In this section, we analyze the complexity of RSCALE when its transformations are applied directly to the more general form of the ABD system (1.8). We begin by stepping through the initial few stages of the algorithm (the first block-step is slightly different than subsequent block-steps), and then specify the general block-step and analyze its complexity. Again, since we are concerned only with transformations that contribute to the high-order coefficient in the operation count, we do not consider the right-hand side in this discussion.

Given the general form of the ABD matrix

$$\mathcal{J} = \begin{bmatrix} \mathcal{B}_a & & & & & & \mathcal{B}_b \\ S_0 & T_0 & & & & & \\ & S_1 & T_1 & & & & \\ & & \ddots & \ddots & & & \\ & & & S_{M-3} & T_{M-3} & & \\ & & & & S_{M-2} & T_{M-2} & \\ & & & & & S_{M-1} & T_{M-1} \end{bmatrix} \tag{4.3}$$

RSCALE first performs a column-differencing transformation:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 - S_0 & -T_0 & & & & \\ & S_1 & T_1 - S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & T_{M-3} - S_{M-3} & -T_{M-3} & \\ & & & & S_{M-2} & T_{M-2} - S_{M-2} & -T_{M-2} \\ & & & & & S_{M-1} & T_{M-1} - S_{M-1} \end{bmatrix} \tag{4.4}$$

The bottom block-row is then transformed by multiplying through by $[T_{M-1} - S_{M-1}]^{-1}$

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 - S_0 & -T_0 & & & & \\ & S_1 & T_1 - S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & T_{M-3} - S_{M-3} & -T_{M-3} & \\ & & & & S_{M-2} & T_{M-2} - S_{M-2} & -T_{M-2} \\ & & & & & \boxed{S_{M-1}^{(0)}} & I \end{bmatrix} \tag{4.5}$$

where $S_{M-1}^{(0)} = [T_{M-1} - S_{M-1}]^{-1} S_{M-1}$. The inverse is not formed explicitly; this transformation is implemented by computing the LU-factorization of $[T_{M-1} - S_{M-1}]$, followed by n-solves. The same is true for all subsequent matrix inversions appearing in this discussion.

Next, we eliminate T_{M-2} from block row $M-2$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0-S_0 & -T_0 & & & & \\ & S_1 & T_1-S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & T_{M-3}-S_{M-3} & -T_{M-3} & \\ & & & & S_{M-2} & \boxed{T^{(1)}_{M-2}} & \\ & & & & & S^{(0)}_{M-1} & I \end{bmatrix} \tag{4.6}$$

where $T^{(1)}_{M-2} = T_{M-2} - S_{M-2} + T_{M-2}S^{(0)}_{M-1} = T_{M-2}(I + S^{(0)}_{M-1}) - S_{M-2}$, and multiply block-row $M-2$ through by $[T^{(1)}_{M-2}]^{-1}$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0-S_0 & -T_0 & & & & \\ & S_1 & T_1-S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & T_{M-3}-S_{M-3} & -T_{M-3} & \\ & & & & \boxed{S^{(1)}_{M-2}} & I & \\ & & & & & S^{(0)}_{M-1} & I \end{bmatrix} \tag{4.7}$$

where $S^{(1)}_{M-2} = [T^{(1)}_{M-2}]^{-1} S_{M-2}$. We then "drag" $S^{(0)}_{M-1}$ over by one block-column by subtracting a multiple of block-row $M-2$ from block-row $M-1$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0-S_0 & -T_0 & & & & \\ & S_1 & T_1-S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & T_{M-3}-S_{M-3} & -T_{M-3} & \\ & & & & S^{(1)}_{M-2} & I & \\ & & & & \boxed{S^{(1)}_{M-1}} & & I \end{bmatrix} \tag{4.8}$$

where $S_{M-1}^{(1)} = -S_{M-1}^{(0)} S_{M-2}^{(1)}$. Next, we eliminate T_{M-3} from block row $M-3$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 - S_0 & -T_0 & & & & \\ & S_1 & T_1 - S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3} & \boxed{T_{M-3}^{(2)}} & & \\ & & & & S_{M-2}^{(1)} & I & \\ & & & & S_{M-1}^{(1)} & & I \end{bmatrix} \tag{4.9}$$

where $T_{M-3}^{(2)} = T_{M-3} - S_{M-3} + T_{M-3} S_{M-2}^{(1)} = T_{M-3}(I + S_{M-2}^{(1)}) - S_{M-3}$, and multiply block-row $M-3$ through by $[T_{M-3}^{(2)}]^{-1}$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 - S_0 & -T_0 & & & & \\ & S_1 & T_1 - S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \boxed{S_{M-3}^{(2)}} & I & & \\ & & & & S_{M-2}^{(1)} & I & \\ & & & & S_{M-1}^{(1)} & & I \end{bmatrix} \tag{4.10}$$

where $S_{M-3}^{(2)} = [T_{M-3}^{(2)}]^{-1} S_{M-3}$. We then drag $S_{M-1}^{(1)}$ over another block-column by subtracting a multiple of block-row $M-3$ from block-row $M-1$:

$$\begin{bmatrix} \mathcal{B}_a & -\mathcal{B}_a & & & & & \mathcal{B}_b \\ S_0 & T_0 - S_0 & -T_0 & & & & \\ & S_1 & T_1 - S_1 & -T_1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & S_{M-3}^{(2)} & I & & \\ & & & & S_{M-2}^{(1)} & I & \\ & & & \boxed{S_{M-1}^{(2)}} & & & I \end{bmatrix} \tag{4.11}$$

where $S_{M-1}^{(2)} = -S_{M-1}^{(1)} S_{M-3}^{(2)}$. We continue in this fashion, working upward toward the top block-row of the matrix. In the final stage, the left boundary block is transformed

$$\mathcal{B}_a^{(M)} = \mathcal{B}_a + \mathcal{B}_a S_0^{(M-1)} = \mathcal{B}_a (I + S_0^{(M-1)})$$

In general, at stage k the RSCALE block-step involves three transformations:

$$T_i^{(k)} \longleftarrow T_i(I + S_{i+1}^{(k-1)}) - S_i \tag{4.12}$$

$$S_i^{(k)} \longleftarrow [T_i^{(k)}]^{-1} S_i \tag{4.13}$$

$$S_{M-1}^{(k)} \longleftarrow -S_{M-1}^{(k-1)} S_i^{(k)} \tag{4.14}$$

(4.12) requires n additions for $(I + S_{i+1}^{(k-1)})$, n^3 flops for the matrix multiplication, and n^2 additions for the matrix difference. (4.13) is implemented with an LU-factorization followed by n solves, for a total cost of $(4/3)n^3 + \mathcal{O}(n^2)$ flops. (We include the cost of computing the Gauss transformation multipliers in the $\mathcal{O}(n^2)$ term, although strictly speaking a division is not a flop.) (4.14) requires n^3 flops for the matrix multiplication. The total cost of all computations performed during a general block-step of RSCALE is therefore $(10/3)n^3 + \mathcal{O}(n^2)$ flops.

If the right-block-identity form of the ABD system (1.9) arises naturally in the discretization, the cost drops to $(7/3)n^3 + \mathcal{O}(n^2)$ flops since when $T_i \equiv I \ \forall \ i$ there is no need for the matrix multiplication in (4.12). It is interesting to note that the overall cost to first transform (1.8) to (1.9), and then apply RSCALE to the transformed system, is slightly greater than the cost of applying RSCALE directly to (1.8) as shown above. In order to transform (1.8) to (1.9) we must compute $[T_i]^{-1}S_i, i = 0, \ldots, M-1$ at a cost of $(4/3)n^3 + \mathcal{O}(n^2)$ flops per block-row. Comparing high-order coefficients of the complexity of the two approaches, we see $(4/3)n^3 + (7/3)n^3 > (10/3)n^3$. In addition, the transformation may not even be possible if one or more of the blocks T_i is poorly-conditioned. (Recall that we are able to control the condition of $[T_i^{(k)}]$ in (4.13) with a suitable choice of relaxation parameter σ (§3.3). We do not include σ in our complexity analysis as it affects the $\mathcal{O}(n)$ term of the operation count only.) The code included in Appendix E is designed to handle the general form of the ABD system directly, as shown in (4.4)-(4.11).

Table 4.1 summarizes the operation counts for the three ABD system solvers. We include operation counts for COLROW [Diaz 83] for comparison. In the remaining sections of this chapter, we discuss the results of several experiments in which we assess the performance of the three parallel ABD system solvers. We expect that for problems of suitable dimension (i.e., n suitably large), the relative execution time of the solvers should approximate the ratio of the high-order coefficients of their respective operation counts. Given the coefficients just derived, we predict the RSCALE/*SLF*-QR and RSCALE/*SLF*-LU relative execution times to be approximately 43% and 87%, respectively. These predicted ratios—43% and 87%—are included as benchmarks in timing experiments throughout this chapter. (We set the *SLF*-LU coefficient at its upper bound in this predicted ratio so that any improvement in *SLF*-LU execution time

Table 4.1: Complexity of the factorization and reduction component of a block-step in each of the parallel ABD system solvers, and COLROW. Operations are counted in terms of *flops*—multiplication/addition pairs.

ABD System Solver	Complexity
SLF-QR	$(23/3)n^3 + \mathcal{O}(n^2)$
SLF-LU	$\mathcal{K}\, n^3 + \mathcal{O}(n^2),\ \mathcal{K} \in [7/3, 23/6]$
RSCALE	$(10/3)n^3 + \mathcal{O}(n^2)$
COLROW	$(5/6)n^3 + \mathcal{O}(n^2)$

Table 4.2: Architecture specification for sequential tests.

Architecture			Specificiations			
			processors			memory
acronym	model	vendor	#	speed	type	
BLA	SunBlade 1000	Sun Microsystems	2	600 MHZ	USPARC3	5 Gigabytes

due to reduced fill-in is immediately apparent. Our experiments show that the coefficient tends toward its upper bound in most problems where *SLF*-LU is stable.)

4.2 Sequential Tests

Each of the experiments discussed in this section was run on the SunBlade 1000. Architecture specifications are given in Table 4.2. All code was compiled using the level 3 (extensive) optimization options available with Sun's Fortran 77 compiler. The objective of the experiments is to measure the relative performance of the three ABD system solvers *SLF*-QR, *SLF*-LU, and RSCALE, when run in sequential mode, on a sequential machine.

We consider linear problems only in this section—problems of the form

$$y'(t) = A(t)y(t) + q(t), \quad t \in [t_a, t_b], \quad \mathcal{B}_a y(t_a) + \mathcal{B}_b y(t_b) = g, \tag{4.15}$$

where $y, q, g \in \mathcal{R}^n$, $A \in \mathcal{R}^{n\times n}$, $\mathcal{B}_a \in \mathcal{R}^{n_a \times n}$ and $\mathcal{B}_b \in \mathcal{R}^{n_b \times n}$. In most cases, we use separable boundary conditions so that the ABD system arising from the discretization may be solved with a band solver, providing an additional means for checking solution accuracy.

Each problem is constructed in such a way that the analytic (or true) solution is known in advance. In particular, if we set

$$q(t) = (I - A(t)) \begin{bmatrix} e^t \\ \vdots \\ e^t \end{bmatrix},$$

the analytic solution is

$$y(t) = [e^t, \ldots, e^t]^T.$$

Once constructed, a problem is discretized using a trapezoidal or mid-point finite-difference discretization on meshes of varying size, and the ABD linear systems that arise are solved with each of the ABD system solvers to yield discrete computed solutions $\tilde{Y}$.

The accuracy of a computed solution $\tilde{Y}$ is measured as the *algebraic error* $\|\tilde{Y} - Y_b\|_\infty$, where Y_b is the solution obtained with LAPACK's band-solver routines DGBTRF/DGBTRS. A condition number estimate for the ABD matrix is computed with DGBCON. The condition number is used to gauge the expected number of correct digits in the band-solver solution, which must be known in order to correctly interpret the algebraic error. The *discretization error* is estimated as $\|Y_b - Y_t\|_\infty$, where Y_t is the discrete analytic solution. In the context of solving a BVODE, a computed solution $\tilde{Y}$ is acceptable if its algebraic error is smaller in magnitude than the discretization error. In most experiments in this section, algebraic errors are several orders of magnitude smaller than the discretization error. Thus, small variations in the algebraic error among the solvers likely are of no concern. Also, since the algebraic error is measured with respect to LAPACK's DGBTRF-DGBTRS, the ABD solver with the "smallest" algebraic error simply agrees best with that particular code. While this is reassuring, it does not imply that the band-solver computes the true algebraic solution. In other words, the algebraic errors listed may not be completely accurate.

Execution time is measured in two ways—*absolute* and *relative*. The absolute execution time required to compute a given solution $\tilde{Y}$ is the time required to complete a call to the ABD system solver as measured by Fortran 77's DTIME. (A call to DTIME is placed immediately before and after the call to the ABD system solver.) In order to avoid possible timing inconsistencies due to swapping in a time-sharing environment, experiments either were run when the machine was quiet (no other users), or results were averaged over several runs. The relative execution time is measured as the ratio of the RSCALE/*SLF*-LU and RSCALE/*SLF*-QR absolute execution times. In each experiment, the relative execution times are compared to their expected values as predicted by the ratios of the high-order (n^3) coefficients of the algorithm

Table 4.3: Nine constant-coefficient linear problem classes to test the sequential performance of *SLF*-QR, *SLF*-LU and RSCALE.

Problems are randomly-generated in each class. Each problem is discretized using trapezoidal finite differences and solved on a single partition. Longer execution time is induced by increasing M, the number of mesh subintervals. The length of the interval of integration $[t_a, t_b]$ is adjusted accordingly in order to maintain a uniform mesh spacing of $h = 20/512$.

Class	n	Structure of A	$\|A\|_\infty$
A	10	two 5^{th} order eqns.	≤ 100
B	10	five 2^{nd} order eqns.	≤ 100
C	12	two 6^{th} order eqns.	≤ 120
D	12	six 2^{nd} order eqns.	≤ 120
E	14	two 7^{th} order eqns.	≤ 140
F	14	seven 2^{nd} order eqns.	≤ 140
K	10	random sparsity	≤ 100
L	12	random sparsity	≤ 120
M	14	random sparsity	≤ 140

Meshes used for each problem		
M	$[t_a, t_b]$	h
512	$[-10.0, 10.0]$	$20/512$
1024	$[-20.0, 20.0]$	$20/512$
2048	$[-40.0, 40.0]$	$20/512$
4096	$[-80.0, 80.0]$	$20/512$

operation counts (§4.1). We use the worst-case coefficient for *SLF*-LU in these ratios, so that any improvement in *SLF*-LU execution time due to reduced fill-in is immediately apparent.

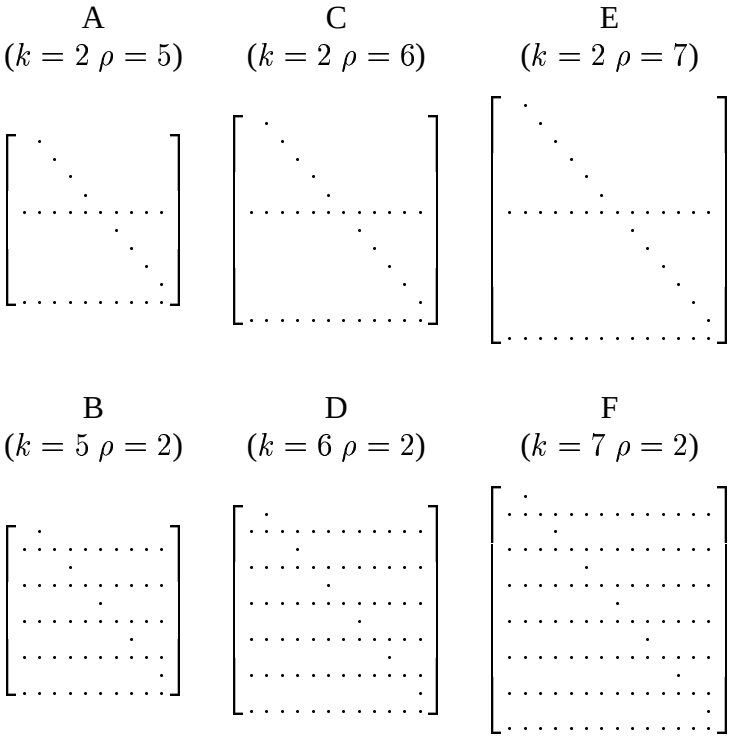
4.2.1 Constant-Coefficient Linear Problems

Table 4.3 lists nine problem classes. Several problems are generated from each class and solved with the ABD system solvers. The problems in classes A-M are all *constant-coefficient*; i.e., they are of the form (4.15) but with $A(t) \equiv A \ \forall \ t$. Variable-coefficient linear problems are considered in §4.2.2, and nonlinear problems are considered in §4.4.

The problems in classes A-F of Table 4.3 are *structured*. The structure of the Jacobian $\partial f / \partial y$ arises from the standard technique of converting a system of higher-order equations into a system of first-order equations. Such a conversion is often necessary before interfacing to mathematical software for solving systems of differential equations. Figure 4.1 shows the Jacobian sparsity pattern. We note that problems in classes A, C, and E have sparse Jacobians,

Figure 4.1: Jacobian structures of the test problems in Table 4.3.

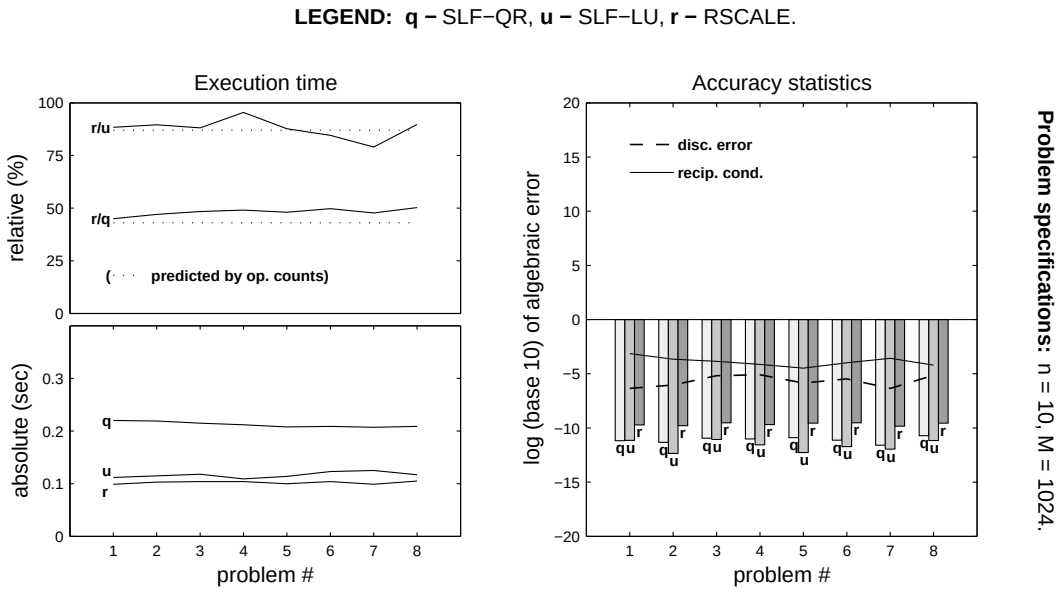
For problems in classes A-F, the structure of $A \equiv \left[\frac{\partial f}{\partial y}\right]$ depicted below arises from converting a system of k ρ-order equations into a system of $n = k\rho$ first-order equations. In each structure, when row i is sparse the only non-zero entry is $a_{i,i+1} = 1$. When row i is dense, $|a_{i,j}| \leq 10, 1 \leq j \leq n$.



whereas problems in classes B, D, and F have denser Jacobians. Problems in classes K, L and M have random structure—the nonzeros of the Jacobian are randomly-distributed throughout the matrix, with at least one nonzero in each row and column.

A given problem in a class is constructed by randomly-generating the nonzeros of the Jacobian. For the structured problems in classes A-F, only elements in the dense rows are randomly-generated. The nonzero in each sparse row is 1. For problems in classes K, L and M, all nonzeros are randomly generated (random in both value and position). Problems generated are of sufficient order ($n = 10, 12$ or 14) that the random generation of Jacobian elements usually results in a strong dichotomy with at least one rapidly increasing and one rapidly decreasing fundamental solution mode. The separable boundary conditions are set up in such a way as to control the solution modes, so that the resulting problem is well-posed. Once constructed, a

Figure 4.2: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class A problems of Table 4.3. Jacobian structure arises from the conversion of two 5^{th} order equations.

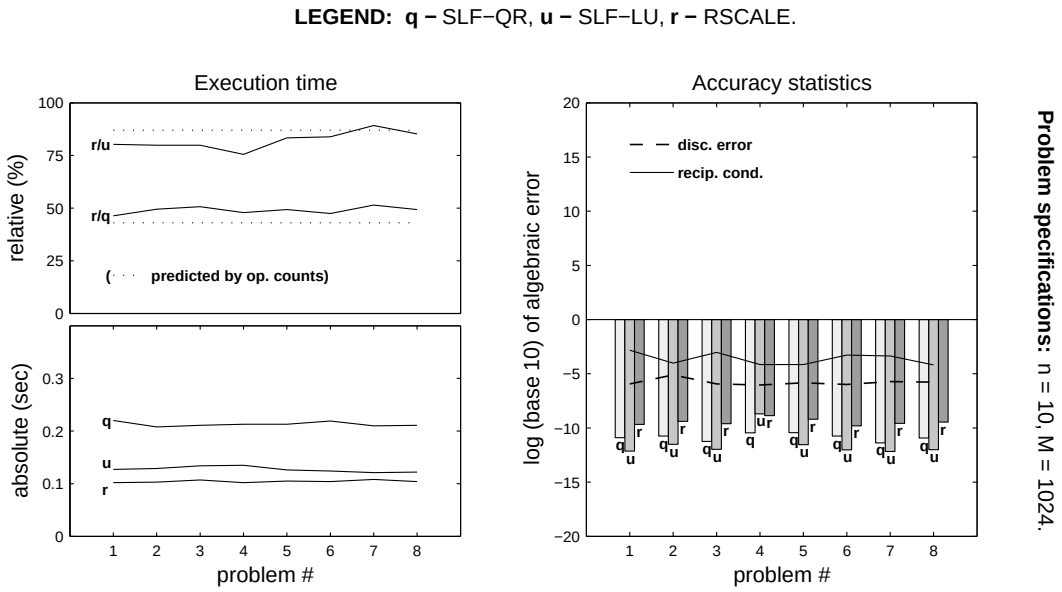


problem is discretized using a trapezoidal finite-difference discretization and the ABD linear system that arises is solved with each of the ABD system solvers.

We now discuss the results of several experiments run on test problems from Table 4.3. These results are summarized in Figures 4.2-4.7 of this section, and in Figures B.5-B.10 of Appendix B. Figure 4.2 shows the execution time and accuracy of the three ABD solvers when solving eight randomly-generated Class A problems. The graph on the right shows that each solver is acceptably accurate on all problems, given that the algebraic errors are all several orders of magnitude less than the discretization error. The algebraic error in the *SLF*-LU solutions is smallest, indicating that *SLF*-LU agrees best with the band-solver in these experiments. The graph on the left shows that RSCALE is approximately 1.2 and 2.2 times faster than *SLF*-LU and *SLF*-QR, respectively, which is in close agreement with the ratios predicted by the operation counts (dotted line). We see slight fluctuations in the RSCALE/*SLF*-LU execution time ratio—in particular, in problem #4. This is an example of reduced *SLF*-LU execution time due to reduced fill-in.

Figure 4.3 shows the execution time and accuracy of the three ABD solvers when solving eight randomly-generated Class B problems. The Jacobians in Class A and Class B problems are of the same dimension ($n = 10$)—they differ only in structure. More precisely, Class B

Figure 4.3: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class B problems of Table 4.3. Jacobian structure arises from the conversion of five 2^{nd} order equations.



problems have denser Jacobians (Figure 4.1). We expect *SLF*-LU execution time to tend toward its upper bound in these problems, and indeed the RSCALE/*SLF*-LU execution time ratios tend to be smaller in the problems in Figure 4.3. This trend is also reflected in a slight increase in displacement between the *SLF*-LU and RSCALE absolute execution times. Again, each solver is acceptably accurate on all problems. Note that the *SLF*-LU solution does not exhibit the smallest algebraic error in problem #4. We will see more examples of the degradation in *SLF*-LU accuracy in subsequent experiments.

Figures 4.4 and 4.5 show experimental results for randomly-generated Class C and Class D problems, respectively. Jacobians are all of dimension $n = 12$—sparse in Class C and dense in Class D. We see the same trend as before; namely, the *SLF*-LU execution time tends toward its upper bound when the Jacobian is denser. There are also more instances of reduced *SLF*-LU accuracy in these problems.

Figures 4.6 and 4.7 show experimental results for randomly-generated Class E and Class F problems, respectively. Jacobians are all of dimension $n = 14$—sparse in Class E and dense in Class F. We see more evidence of the sensitivity of *SLF*-LU execution time to Jacobian density, and two dramatic examples of *SLF*-LU instability. In the problems where *SLF*-LU fails (#4 and #8 of Figure 4.6) the RSCALE/*SLF*-LU execution time ratio is close to 100%. In fact, the

Figure 4.4: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class C problems of Table 4.3. Jacobian structure arises from the conversion of two 6^{th} order equations.

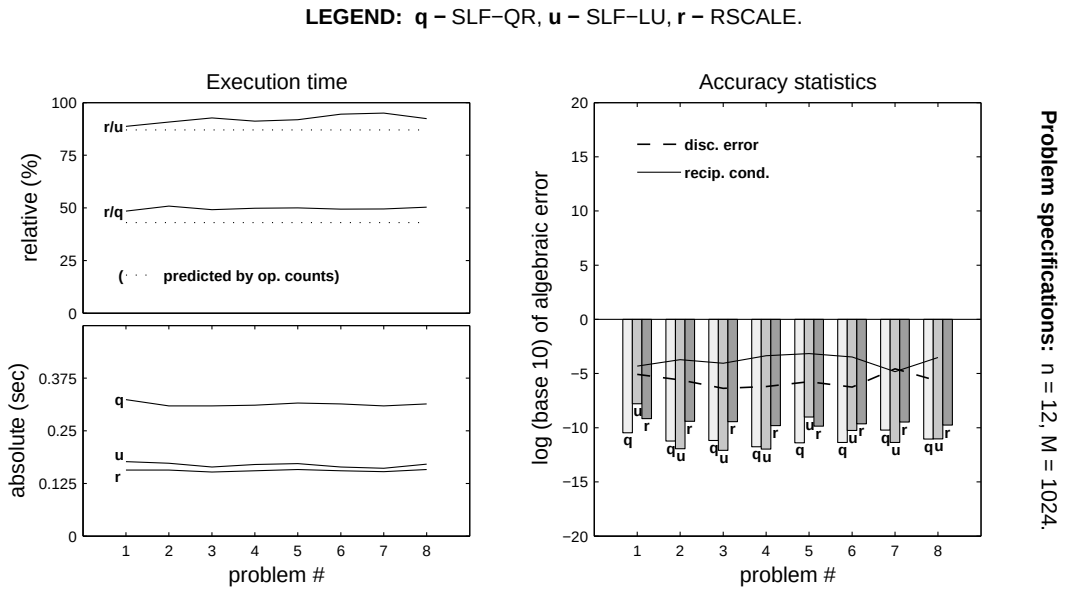


Figure 4.5: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class D problems of Table 4.3. Jacobian structure arises from the conversion of six 2^{nd} order equations.

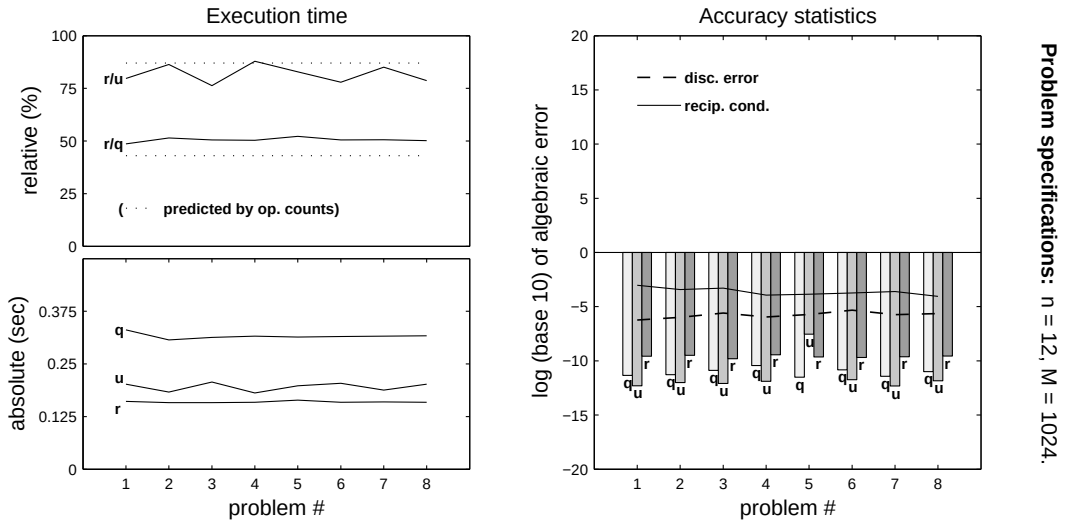
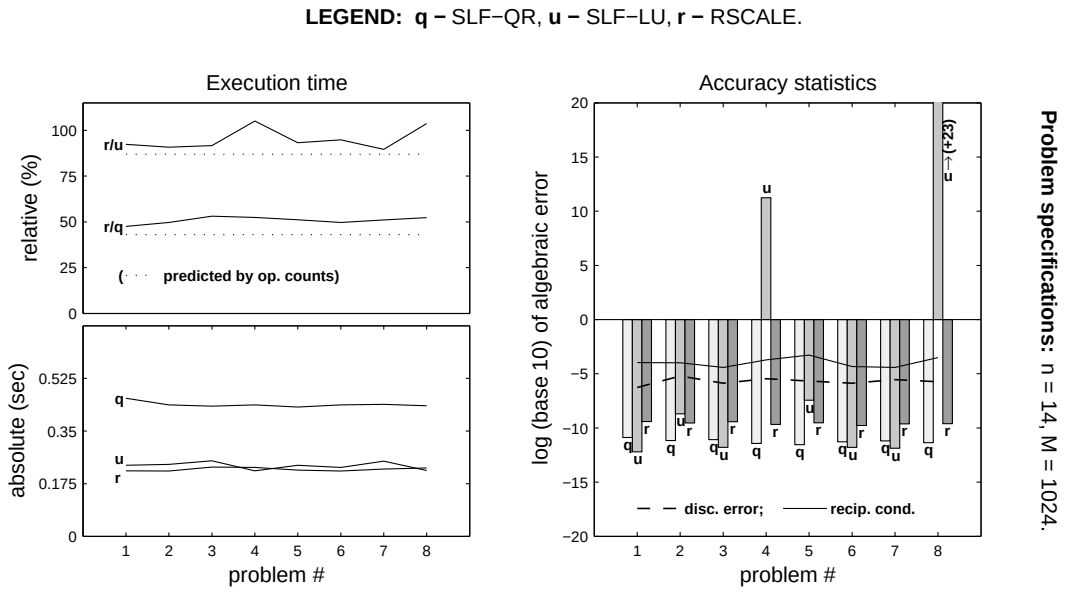


Figure 4.6: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class E problems of Table 4.3. Jacobian structure arises from the conversion of two 7^{th} order equations.



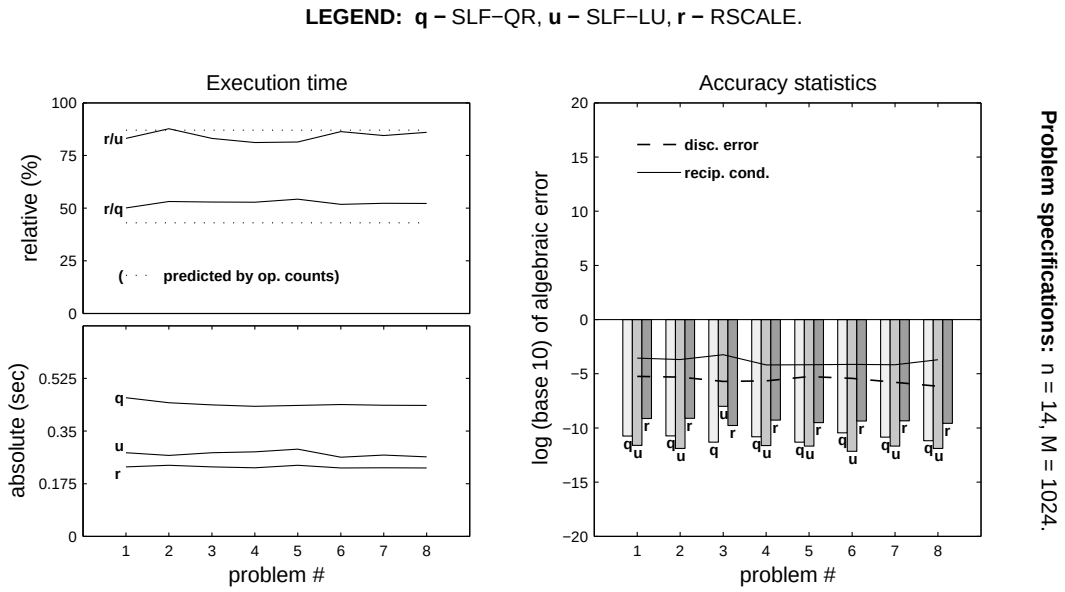
ratio is actually slightly greater than 100% indicating that *SLF*-LU is faster on these problems than RSCALE. Closer investigation shows that *SLF*-LU does not pivot at all on these problems, explaining both its improved speed and degraded stability. (When *SLF*-LU does not pivot, it is equivalent to unstable compactification. See §2.1 and §3.2.)

Problems in classes K, L and M exhibit random Jacobian structure. While these problems are not likely to arise in practice, they are somewhat useful for investigating the pivoting and fill-in behaviour of *SLF*-LU. Experimental results for sixteen randomly-generated Class K problems ($n = 10$) are shown in Figures B.5 and B.6 of Appendix B. As Jacobian density is increased from 20% to 90% nonzero, we see a notable trend toward increased *SLF*-LU execution time. Similar trends are evident in Figures B.7 and B.8 (Class L problems, $n = 12$) and Figures B.9 and B.10 (Class M problems, $n = 14$). Note the problems where *SLF*-LU exhibits instability in Figures B.6, B.7, B.8 and B.9.

4.2.2 Variable-Coefficient Linear Problems

Several model variable-coefficient linear problems have appeared in the literature for testing BVODE codes and ABD solvers. We have selected four of these problems for the experiments in this section. For each problem, we perform two experiments. In the first, we solve the

Figure 4.7: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class F problems of Table 4.3. Jacobian structure arises from the conversion of seven 2^{nd} order equations.



problem on meshes of increasing number of subintervals, using the single partition variant of each solver. In the second, we solve the problem on a fixed mesh and vary the number of partitions used by each solver. The second experiment is intended to gauge the sensitivity of the solvers to the partitioning strategy. Ideally the partitioning strategy should have no effect on the computed solution, but this is not always the case. (See, for example, Figures 4.61 and 4.62 in §4.4.3.) In all experiments, the accuracy and efficiency of the computed solutions is analyzed and compared as in §4.2.1, using the criteria outlined at the beginning of §4.2.

Problem R (Ascher and Chan [Asch 91]) $a = 0, b = 1, n = 2,$

$$A(t) = \begin{bmatrix} -\lambda \cos 2\omega t & \omega + \lambda \sin 2\omega t \\ -\omega + \lambda \sin 2\omega t & \lambda \cos 2\omega t \end{bmatrix}, \quad q(t) = (I - A(t)) \begin{bmatrix} e^t \\ e^t \end{bmatrix},$$

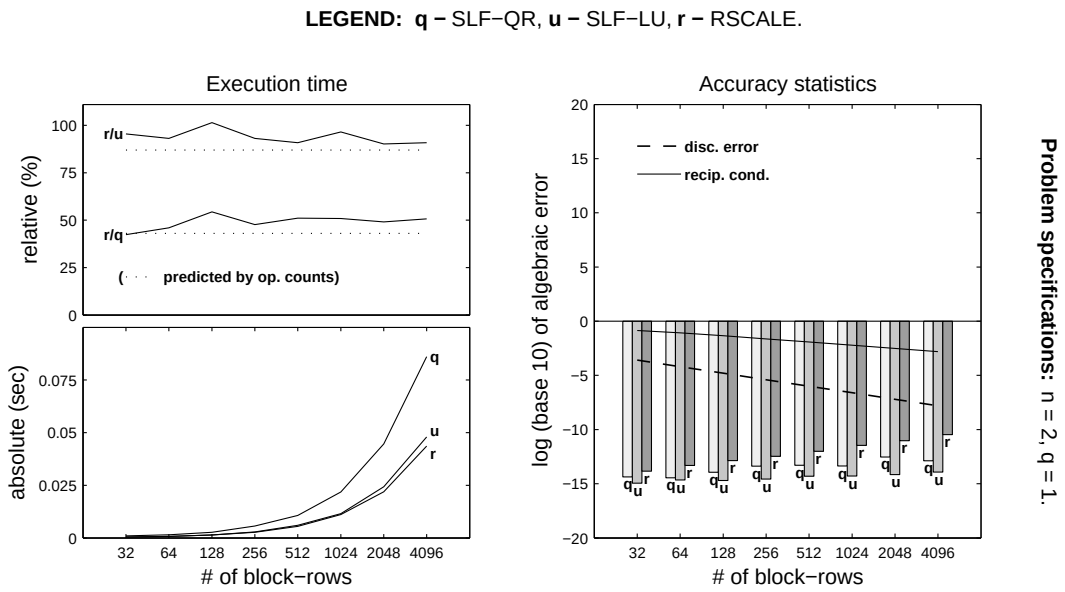
subject to the boundary conditions $y_1(0) = 1,\ y_1(1) = e$. The true solution is $y(t) = e^t(1, 1)^T$.

A fundamental solution matrix is

$$Y(t) = \begin{bmatrix} \cos \omega t & \sin \omega t \\ -\sin \omega t & \cos \omega t \end{bmatrix} \begin{bmatrix} e^{-\lambda t} & 0 \\ 0 & e^{\lambda t} \end{bmatrix},$$

so the dichotomy is clear: there is one decaying and one growing mode. The problem is first solved with $\lambda = 10, \omega = 1$ (Figures 4.8 and 4.9), and then with $\lambda = 200, \omega = 1$ (Figures 4.10

Figure 4.8: Execution time and accuracy of the three sequential ABD solvers when solving Problem R (Ascher-Chan) with $\lambda = 10, \omega = 1$. Solved on one partition, with M varying from 32 to 4096 block-rows.



and 4.11). With $\lambda = 10$ the fundamental solution modes are "slow" and the problem is not difficult numerically. *SLF*-LU likely does not pivot at all on this problem which should result in reduced execution time. However, the order of the Jacobian is too small for this effect to be noticeable. With $\lambda = 200$ the modes are "fast" and the problem is difficult numerically. This case was one of the earliest problems for testing parallel ABD solvers. We see that in all four experiments with Problem R, each solver is acceptably accurate on all problems and the relative execution times are in close agreement with the predicted ratios. Partitioning in Figures 4.9 and 4.11 has little effect on either accuracy or execution time.

Problem S (Mattheij [Matt 85]) $a = 0, b = \pi, n = 3,$

$$A(t) = \begin{bmatrix} 1 - 19\cos 2t & 0 & 1 + 19\sin 2t \\ 0 & 19 & 0 \\ -1 + 19\sin 2t & 0 & 1 + 19\cos 2t \end{bmatrix}, \quad q(t) = (I - A(t)) \begin{bmatrix} e^t \\ e^t \\ e^t \end{bmatrix},$$

subject to the boundary conditions

$$\begin{aligned} y_1(0) &= 1 \\ y_2(\pi) &= e^\pi \\ y_1(\pi) + 3y_3(\pi) &= 4e^\pi. \end{aligned}$$

Figure 4.9: Execution time and accuracy of the three sequential ABD solvers when solving Problem R (Ascher-Chan) with $\lambda = 10, \omega = 1, M = 2048$. The number of partitions is varied.

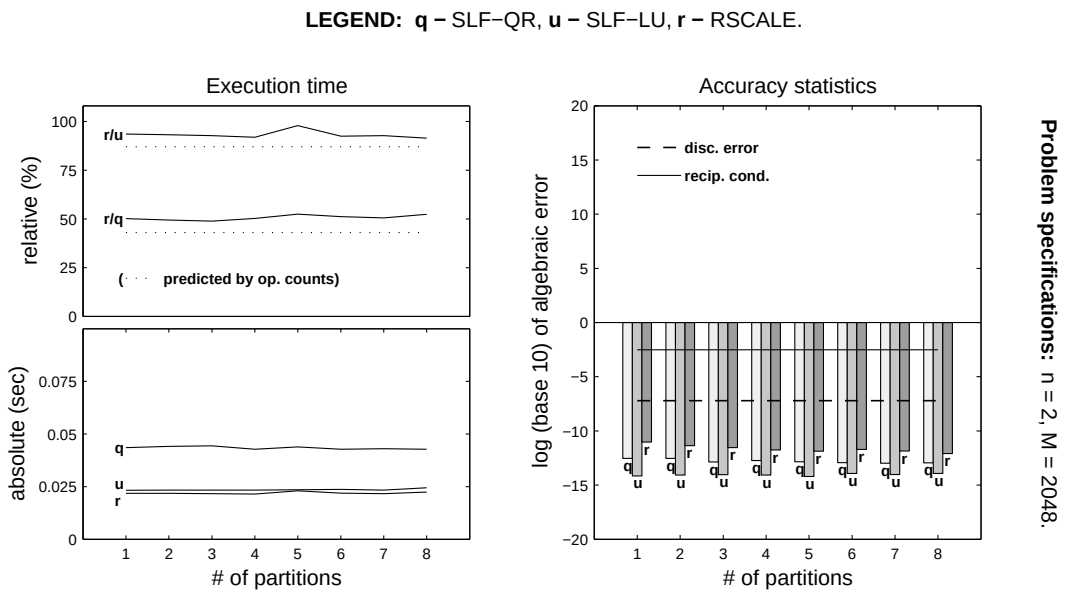


Figure 4.10: Execution time and accuracy of the three sequential ABD solvers when solving Problem R (Ascher-Chan) with $\lambda = 200, \omega = 1$. Solved on one partition, with M varying from 32 to 4096 block-rows.

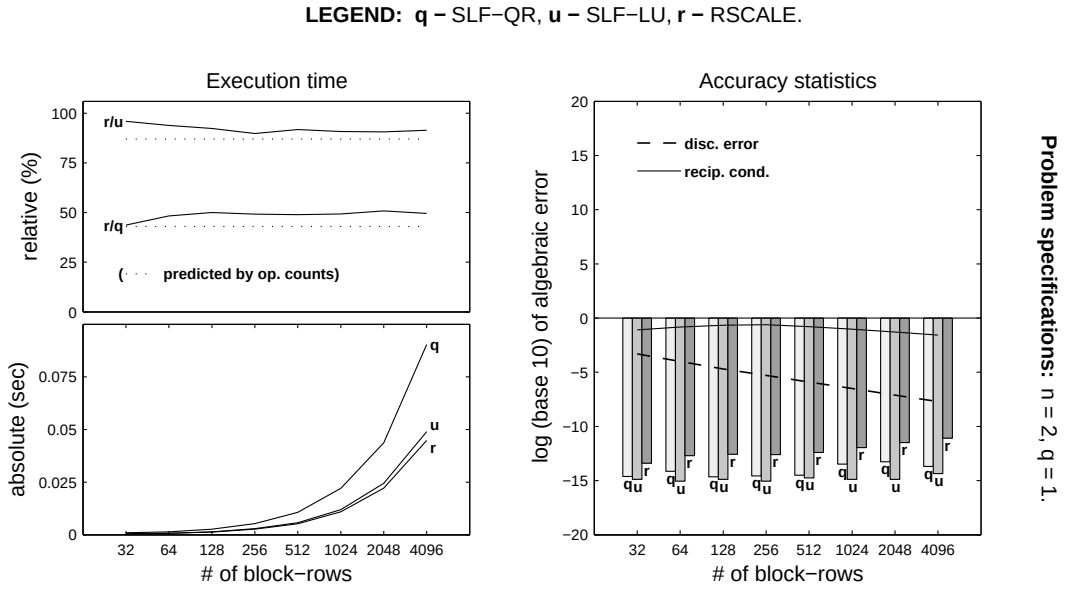
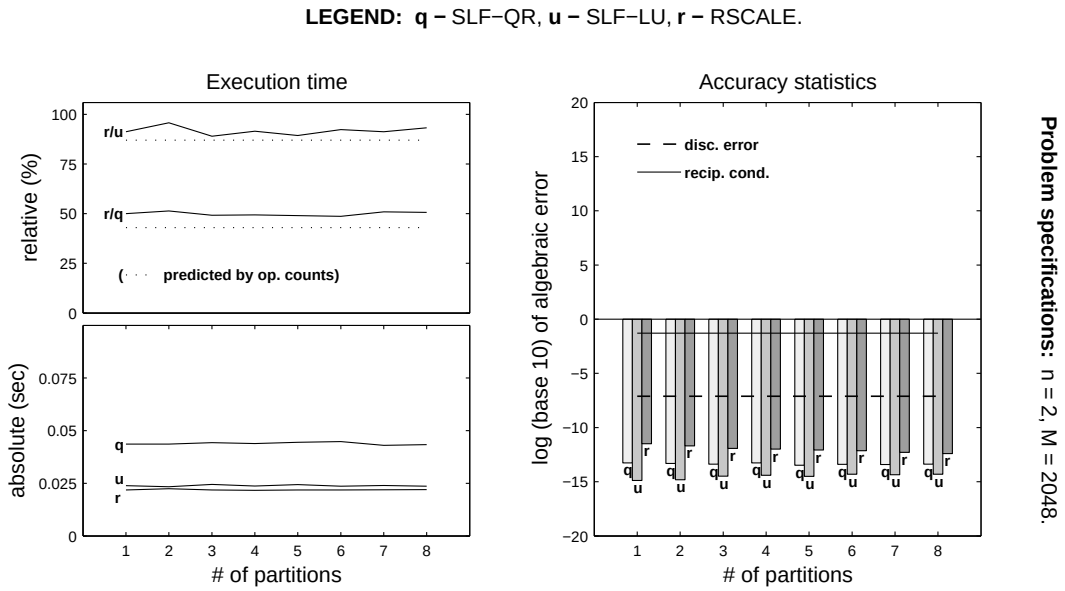


Figure 4.11: Execution time and accuracy of the three sequential ABD solvers when solving Problem R (Ascher-Chan) with $\lambda = 200, \omega = 1, M = 2048$. The number of partitions is varied.



The true solution is $y(t) = e^t(1,1,1)^T$. A fundamental solution matrix is

$$Y(t) = \begin{bmatrix} \sin t & 0 & -\cos t \\ 0 & 1 & 0 \\ \cos t & 0 & \sin t \end{bmatrix} \begin{bmatrix} e^{20t} & 0 & 0 \\ 0 & e^{19t} & 0 \\ 0 & 0 & e^{-18t} \end{bmatrix}.$$

There is one decaying and two growing solution modes, and the modes are fast enough to make the problem difficult numerically. Figures 4.12 and 4.13 show the results of the experiments on this problem. Each solver is acceptably accurate on all problems and the relative execution times are in close agreement with the predicted ratios. (The large fluctuation in the RSCALE/*SLF*-LU execution time ratio in Figure 4.12 is likely an anomaly, although precautions were taken to avoid timing inconsistencies.) Figure 4.13 shows that partitioning has only a minor effect on the accuracy of the solvers.

Problem T (Wright [Wrig 94]) $a = 0, b = 1, n = 5,$

$$A(t) = \begin{bmatrix} -\lambda_1 \cos 2\omega_1 t & 0 & \omega_1 + \lambda_1 \sin 2\omega_1 t & 0 & 0 \\ 0 & -\lambda_2 \cos 2\omega_2 t & 0 & \omega_2 + \lambda_2 \sin 2\omega_2 t & 0 \\ -\omega_1 + \lambda_1 \sin 2\omega_1 t & 0 & \lambda_1 \cos 2\omega_1 t & 0 & 0 \\ 0 & -\omega_2 + \lambda_2 \sin 2\omega_2 t & 0 & \lambda_2 \cos 2\omega_2 t & 0 \\ 0 & 0 & 0 & 0 & \lambda_3 \end{bmatrix},$$

Figure 4.12: Execution time and accuracy of the three sequential ABD solvers when solving Problem S (Mattheij). Solved on one partition, with M varying from 32 to 4096 block-rows.

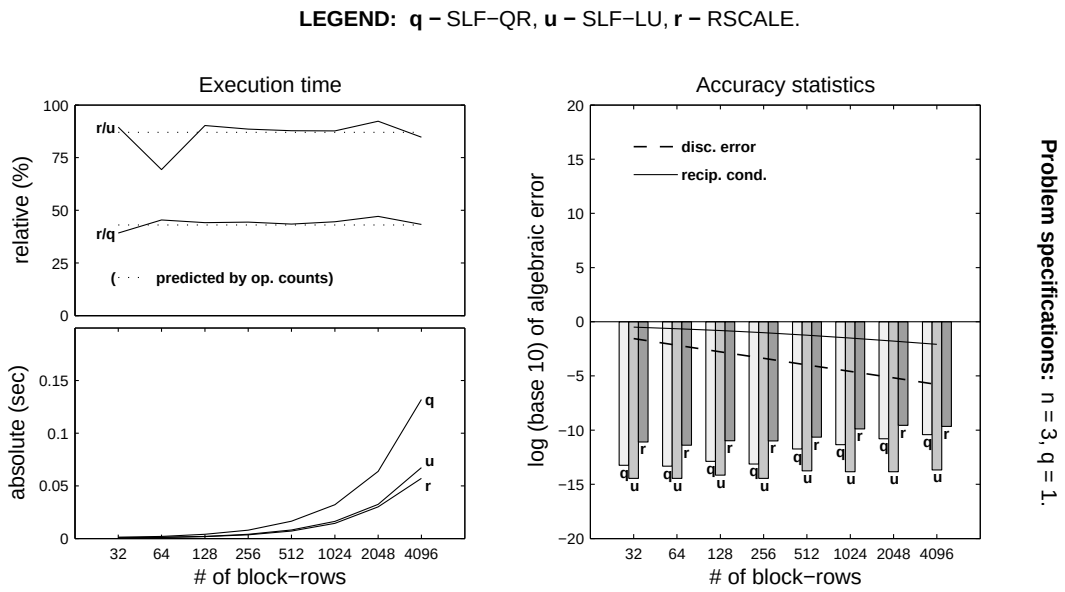
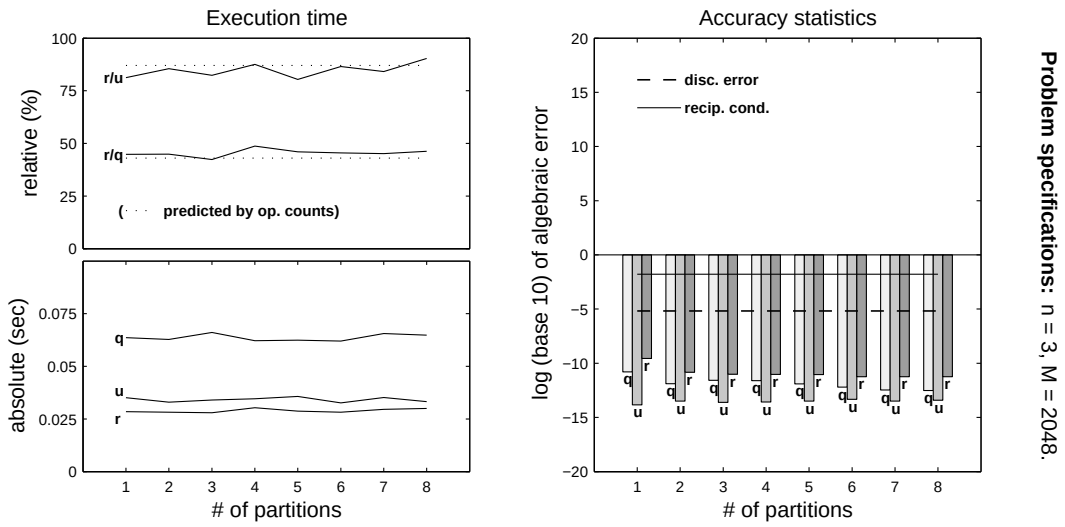


Figure 4.13: Execution time and accuracy of the three sequential ABD solvers when solving Problem S (Mattheij) with $M = 2048$. The number of partitions is varied.



$$q(t) = (I - A(t)) \left[e^t, e^t, e^t, e^t, e^t\right]^T,$$

subject to the boundary conditions

$$\begin{aligned} y_1(0) &= 1 \\ y_2(0) + 4y_5(0) &= 5 \\ y_1(1) &= e \\ -y_3(1) + y_4(1) &= 0 \\ -4y_2(1) + 5y_5(1) &= e. \end{aligned}$$

The true solution is $y(t) = e^t(1, 1, 1, 1, 1)^T$. The problem is solved with $\lambda_1 = 200$, $\lambda_2 = 50$, $\lambda_3 = 10$, $\omega_1 = 1$, and $\omega_2 = 25$. A fundamental solution is

$$Y(t) = \begin{bmatrix} \cos t & 0 & \sin t & 0 & 0 \\ 0 & \cos 25t & 0 & \sin 25t & 0 \\ -\sin t & 0 & \cos t & 0 & 0 \\ 0 & -\sin 25t & 0 & \cos 25t & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} e^{-200t} & 0 & 0 & 0 & 0 \\ 0 & e^{-50t} & 0 & 0 & 0 \\ 0 & 0 & e^{200t} & 0 & 0 \\ 0 & 0 & 0 & e^{50t} & 0 \\ 0 & 0 & 0 & 0 & e^{10t} \end{bmatrix}.$$

There are two decaying and three growing solution modes, and the modes are fast enough to make the problem difficult numerically. Figures 4.14 and 4.15 show the results of the experiments on this problem. Each solver is acceptably accurate on all problems and the relative execution times are in close agreement with the predicted ratios. Figure 4.15 shows that partitioning has only a minor effect on the accuracy of the solvers.

Problem U (Jackson [Asch 91]) $a = 0, b = 10, n = 2,$

$$A(t) = U^T(t) \begin{bmatrix} -1 & 10 \\ 0 & -2 \end{bmatrix} U(t), \quad q(t) = (I - 10A(t)) \begin{bmatrix} e^{t/10} \\ e^{t/10} \end{bmatrix},$$

where

$$U(t) = \begin{bmatrix} \cos 2\pi t & -\sin 2\pi t \\ \sin 2\pi t & \cos 2\pi t \end{bmatrix},$$

subject to the boundary conditions

$$\begin{aligned} y_1(0) &= 10 \\ y_1(10) &= 10e. \end{aligned}$$

The true solution is $y(t) = 10e^{t/10}(1, 1)^T$.

Figure 4.14: Execution time and accuracy of the three sequential ABD solvers when solving Problem T (Wright). Solved on one partition, with M varying from 32 to 4096 block-rows.

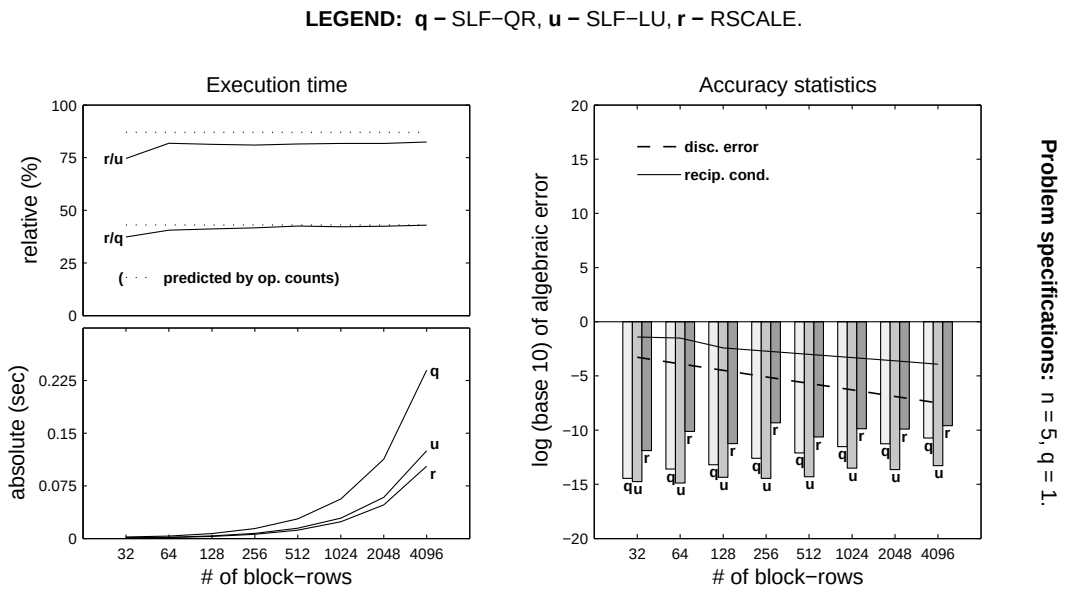


Figure 4.15: Execution time and accuracy of the three sequential ABD solvers when solving Problem T (Wright) with $M = 2048$. The number of partitions is varied.

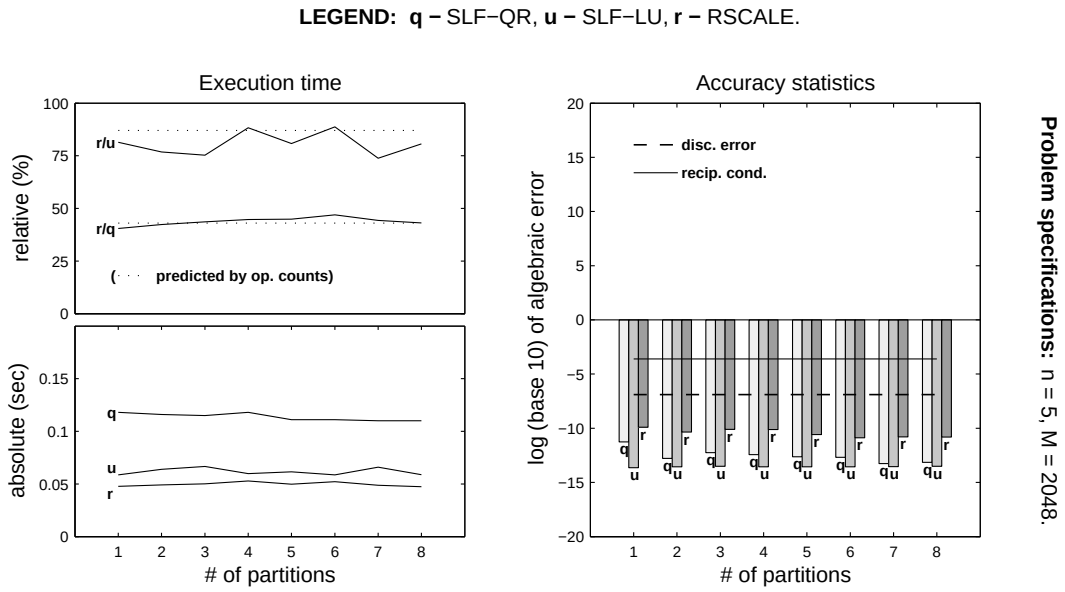
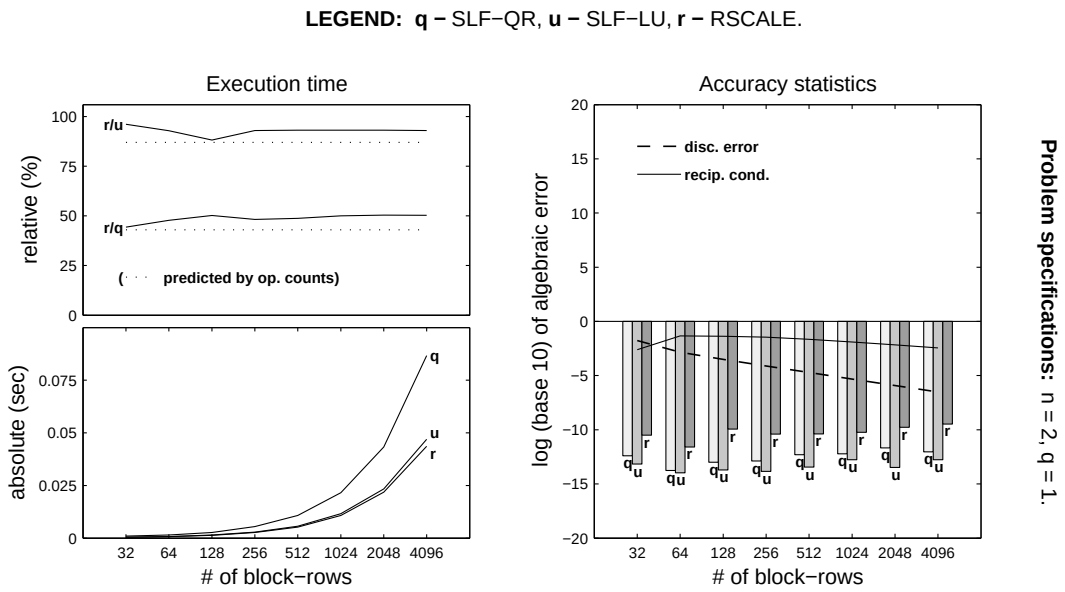


Figure 4.16: Execution time and accuracy of the three sequential ABD solvers when solving Problem U (Jackson). Solved on one partition, with M varying from 32 to 4096 block-rows.



Since $U(t)$ is orthonormal, the eigenvalues of $A(t)$ are $[-1, -2]$ for all t. This problem was chosen as a test case because, in a boundary-value problem, it is not typical for the spectrum of $A(t)$ to be negative for all t. However, the kinematic eigenvalues are 3.3583 and -6.3583, so this ODE has an exponential dichotomy and the problem is well-posed. (See [Asch 88] for a discussion of these concepts.)

Figures 4.16 and 4.17 show the results of the experiments on this problem. Each solver is acceptably accurate on all problems and the relative execution times are in close agreement with the predicted ratios. Figure 4.17 shows that partitioning has little effect on either accuracy or execution time.

4.3 Parallel Tests

Each of the experiments discussed in this section was run on the Origin 2000. Architecture specifications are given in Table 4.4. The Origin 2000 is a tightly-coupled, shared-memory machine. All code was compiled using the level 3 (extensive) optimization options available with the Silicon Graphics Fortran 77 compiler. The objective of the experiments is to measure the relative performance of the three ABD system solvers *SLF*-QR, *SLF*-LU, and RSCALE, when run in parallel mode, on a parallel machine.

Figure 4.17: Execution time and accuracy of the three sequential ABD solvers when solving Problem U (Jackson) with $M = 2048$. The number of partitions is varied.

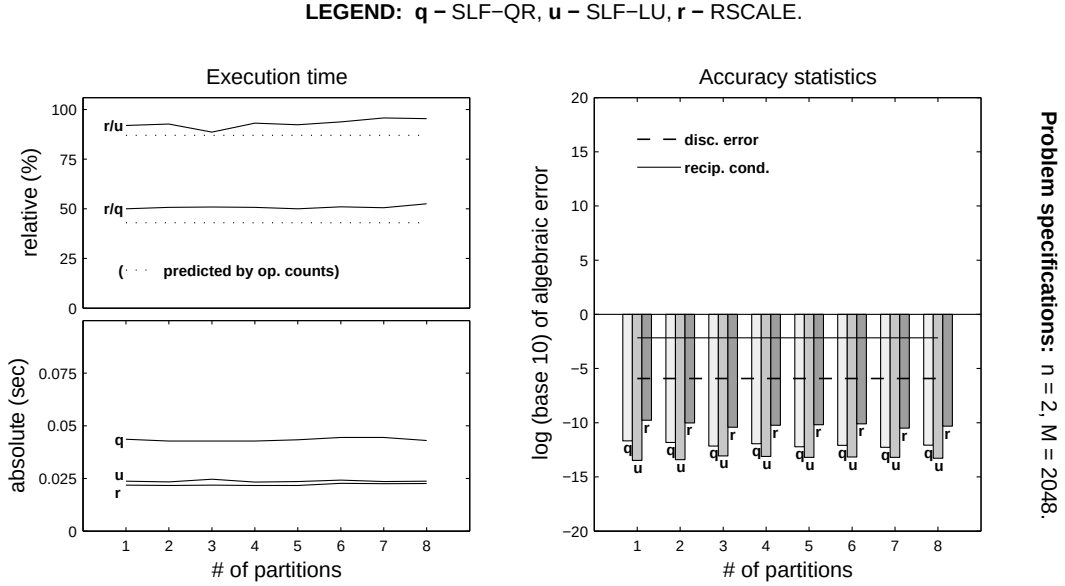


Table 4.4: Architecture specification for parallel tests.

Architecture			Specificiations			
			processors			memory
acronym	model	vendor	#	speed	type	
ORG	Origin 2000	Silcon Graphics	8	250 MHZ	R10000	2 Gigabytes

4.3.1 Compiler Directives

Parallelism and memory partitioning on the Origin 2000 is achieved by using shared-memory parallel compiler directives provided by the Silicon Graphics Fortran 77 compiler. The partitioning algorithms discussed in §2.2 and §2.3 readily translate into code with loop iterations that can be performed independently, and it is straightforward to prepare such loops for parallel execution.

As an example, consider the following simple loop for processing a one-dimensional array:

```
      integer j, M
      real Y(1024)

      M = 1024
      do 10 j = 1, M
          Y(j) = 0.0d0
 10   continue
```

Clearly, each iteration of this loop could be performed independently and in random order without affecting the outcome. We could also group the iterations into `num_partition` slices of `M/num_partition` iterations each:

```
      integer j, k, M, num_partition, partition_size
      real Y(1024)

      M = 1024
      num_partition = 16
      partition_size = M/num_partition
      do 20 k = 1, num_partition
          do 10 j = (k-1)*partition_size + 1, k*partition_size
              Y(j) = 0.0d0
 10       continue
 20   continue
```

and each *group* of iterations could be performed independently and in random order without affecting the outcome. (This, of course, is the partitioning strategy used in all partitioning algorithms discussed in §2.2 and §2.3.) The partitioned loop is designated for parallel execution by inserting a `$DOACROSS` compiler directive as follows:

```
      integer j, k, M, num_partition, partition_size
      real Y(1024)

      M = 1024
      num_partition = 16
      partition_size = M/num_partition
C$DOACROSS SHARE (Y, num_partition, partition_size),
C$&        LOCAL (j, k)
      do 20 k = 1, num_partition
          do 10 j = (k-1)*partition_size + 1, k*partition_size
              Y(j) = 0.0d0
  10      continue
  20  continue
```

Variables (memory locations) specified in the SHARE list may be accessed concurrently by all processors; variables in the LOCAL list are private to each processor. The directive allows each iteration of the outer loop to be performed independently and concurrently on its own processor if enough processors are available.

The original simple loop could also have been parallelized with a $DOACROSS directive, but the random concurrent access of single elements in Y by several processors would result in a degradation of parallel performance. In the partitioned approach, each processor accesses a contiguous slice of Y of size partition_size, which permits more efficient memory management when input and output is buffered. For similar reasons, care must be taken to avoid memory access collisions when sharing small arrays (such as workspace arrays). Partitions must be spread apart sufficiently so that a processor does not lock (or rarely locks) memory locations in adjacent partitions during buffered output.

Many more examples of the use of the $DOACROSS directive may be found in the code included in Appendix E.

4.3.2 Test Problems and Numerical Results

Six test problems are specified in Table 4.5. Each problem is linear and constant-coefficient—as defined in §4.2.1—with a structured Jacobian. Figure 4.18 shows the Jacobian sparsity patterns. The elements in the dense rows of the Jacobian are randomly-generated, and the nonzero in each sparse row is 1. Further details about this type of Jacobian structure are given in §4.2.1.

Table 4.5: Six constant-coefficient linear problems to test the parallel performance of *SLF*-QR, *SLF*-LU and RSCALE.

Each problem is discretized using trapezoidal finite differences. Longer execution time is induced by increasing M, the number of mesh subintervals. The length of the interval of integration $[t_a, t_b]$ is adjusted accordingly in order to maintain a uniform mesh spacing of $h = 25/512$.

Problem #	n	Structure of A	$\|A\|_\infty$
A	6	two 3^{rd} order eqns.	≤ 600
B	8	four 2^{nd} order eqns.	≤ 800
C	10	two 5^{th} order eqns.	≤ 1000
D	12	six 2^{nd} order eqns.	≤ 1200
E	14	seven 2^{nd} order eqns.	≤ 1400
F	16	four 4^{th} order eqns.	≤ 1600

Meshes used for each problem		
M	$[t_a, t_b]$	h
512	$[-12.5, 12.5]$	$25/512$
1024	$[-25.0, 25.0]$	$25/512$
2048	$[-50.0, 50.0]$	$25/512$
4096	$[-100.0, 100.0]$	$25/512$

Once constructed, a problem is discretized using a trapezoidal finite-difference discretization on meshes of varying size, and the ABD linear systems that arise are solved with each of the parallel ABD system solvers.

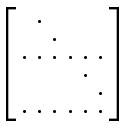
As with the sequential tests in §4.2, we measure both absolute and relative execution time in the parallel tests, showing the RSCALE/*SLF*-LU and RSCALE/*SLF*-QR relative times and comparing them to their expected values as predicted by the ratios of the high-order coefficients of the algorithm operation counts. In each experiment, a given problem is solved and re-solved on an increasing number of processors (1, 2, ..., 8), using an increasing number of partitions (the number of partitions is set to match the number of processors), and the execution times are plotted. We expect to see a decrease in execution time as the number of processors increases. We say that *optimal speedup* is achieved if the execution time on p processors is p times faster than the execution time on 1 processor. As it is not immediately obvious if a code is achieving optimal (or close to optimal) speedup by looking only at its absolute execution time, we use a *speedup graph* to determine this measure. A speedup graph is simply a plot of t_1/t_p versus p, where t_1 and t_p are the absolute execution times on 1 and p processors, respectively. As p increases, the closer t_1/t_p is to p, the closer we are to achieving optimal speedup.

Another useful measure of the gain of parallelism is the number of processors required

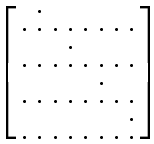
Figure 4.18: Jacobian structures of the test problems in Table 4.5.

The structures of $A \equiv \left[\frac{\partial f}{\partial y}\right]$ depicted below arise from converting a system of k ρ-order equations into a system of $n = k\rho$ first-order equations. In each structure, when row i is sparse the only non-zero entry is $a_{i,i+1} = 1$. When row i is dense, $|a_{i,j}| \leq 100, 1 \leq j \leq n$.

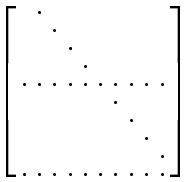
A
($k = 2$ $\rho = 3$)



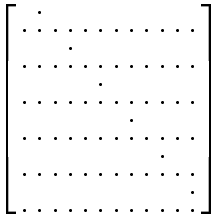
B
($k = 4$ $\rho = 2$)



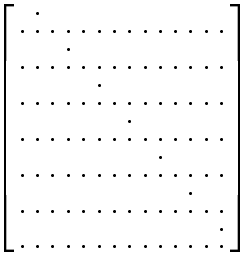
C
($k = 2$ $\rho = 5$)



D
($k = 6$ $\rho = 2$)



E
($k = 7$ $\rho = 2$)



F
($k = 4$ $\rho = 4$)

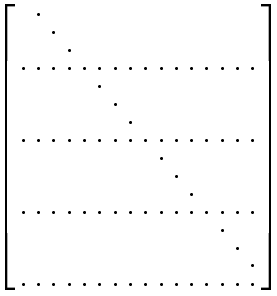
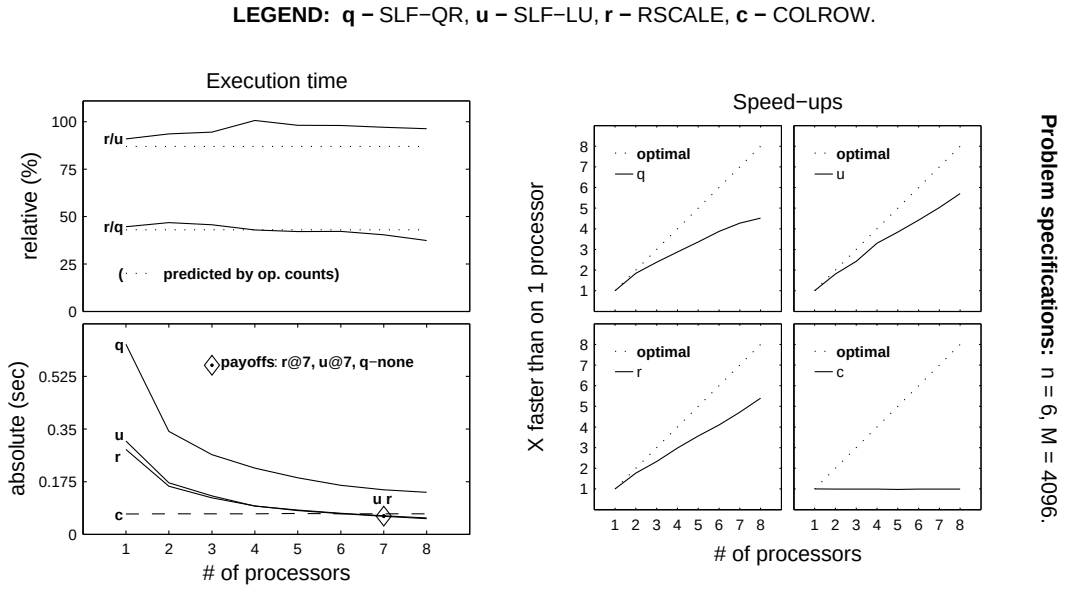


Figure 4.19: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem A of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 7 processors; there is no payoff when using *SLF*-QR.

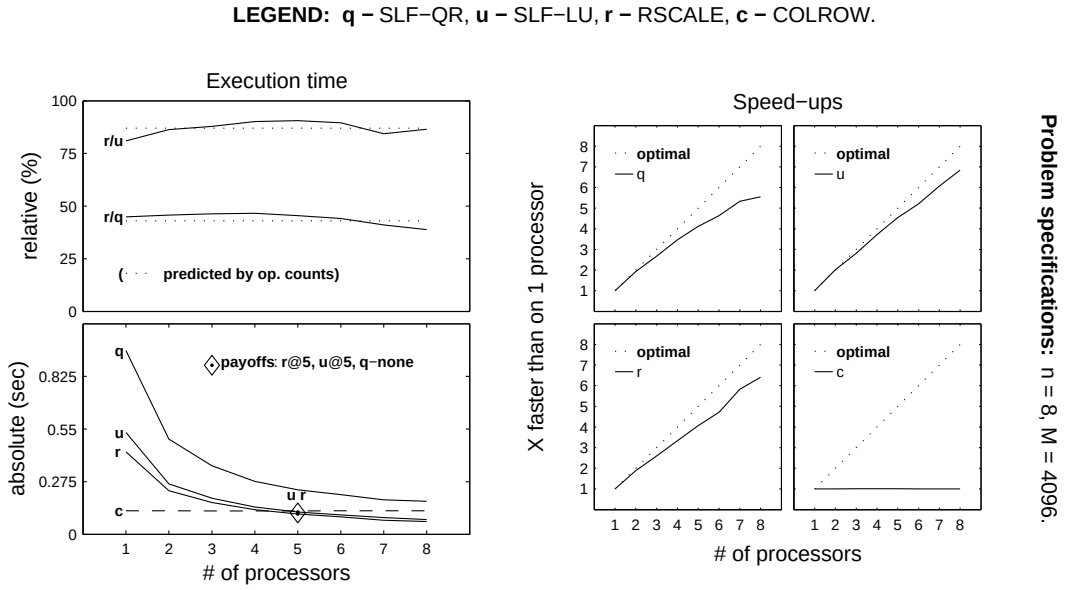


before a parallel code outperforms the best sequential code for the problem. To this end, we solve each of the problems with COLROW [Diaz 83], and plot the absolute execution time of the COLROW solution along with the times of the parallel solver solutions. As p increases and the parallel solver times decrease, the COLROW time remains constant. The point at which the times cross is defined as the *payoff*—the number of processors required before the parallel solver can solve the problem faster than COLROW.

We do not report on the accuracy of computed solutions in these experiments. Each solver is acceptably accurate on all problems, with all partitionings. (The algebraic error in a computed solution is always several orders of magnitude less than the discretization error; see §4.2.)

Figure 4.19 shows the execution time and speedup of the three ABD solvers when solving Problem A of Table 4.5, with $M = 4096$. Absolute execution times decrease as the number of processors increases, and the speedup graphs on the right show that good—but less than optimal—speedup is achieved. (Better speedups are achieved in subsequent experiments when the Jacobian is of greater dimension.) The RSCALE/*SLF*-QR execution time ratio is in close agreement with the predicted ratio. The RSCALE/*SLF*-LU ratio is slightly greater than predicted, which is to be expected as the Jacobian in this problem is sparse. RSCALE and *SLF*-LU each payoff at 7 processors; there is no payoff when using *SLF*-QR.

Figure 4.20: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem B of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 5 processors; there is no payoff when using *SLF*-QR.



Figures 4.20-4.24 show execution times and speedups when solving Problem B-F of Table 4.5. $M = 4096$ in all experiments. As the Jacobian dimension increases from $n = 8$ to $n = 16$, we see two trends:

- Speedups become closer to optimal, and
- payoffs occur with fewer processors.

Both of these trends can be attributed to the fact that the local operations performed during the solution process (in particular, the local factorizations) become more computationally intensive as n increases. When execution time is dominated by the cost of these local operations, any time lost to overhead—such as non-parallelizable tasks related to resolving memory access contention—becomes less significant. As overhead becomes less significant, speedup becomes more optimal. And since COLROW is approximately 4 times faster than sequential RSCALE and *SLF*-LU, and approximately 7 times faster than sequential *SLF*-QR, we would expect payoffs to occur at 4 and 7 processors, respectively. When local operations dominate execution time, these payoffs are realized.

Appendix C contains some additional experiments with problems A-F of Table 4.5.

Figure 4.21: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem C of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 4 processors; there is no payoff when using *SLF*-QR.

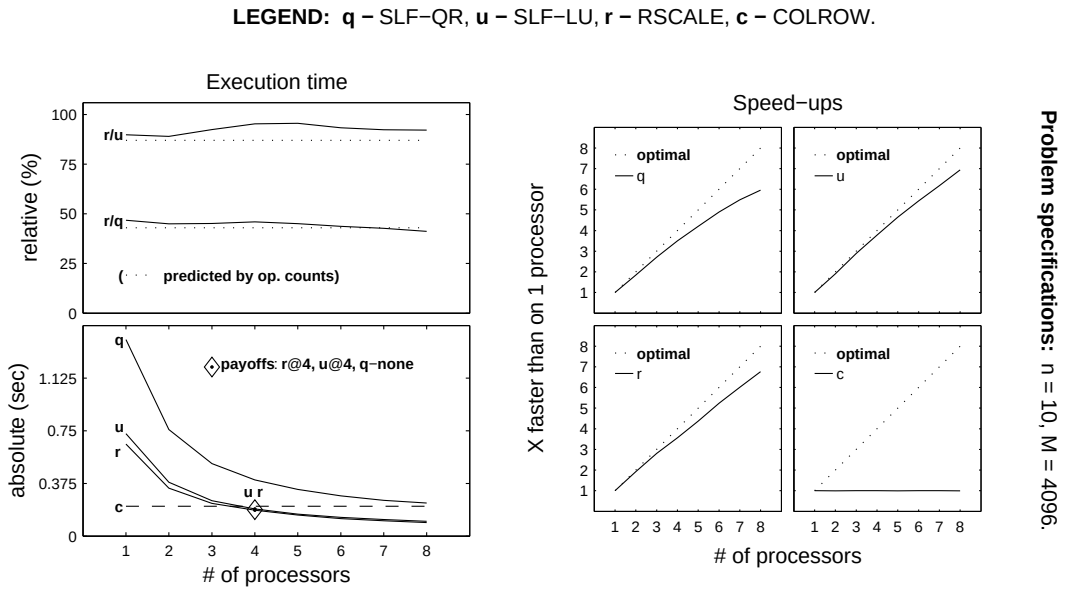


Figure 4.22: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem D of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE, *SLF*-LU and *SLF*-QR payoff at 4, 4 and 8 processors, respectively.

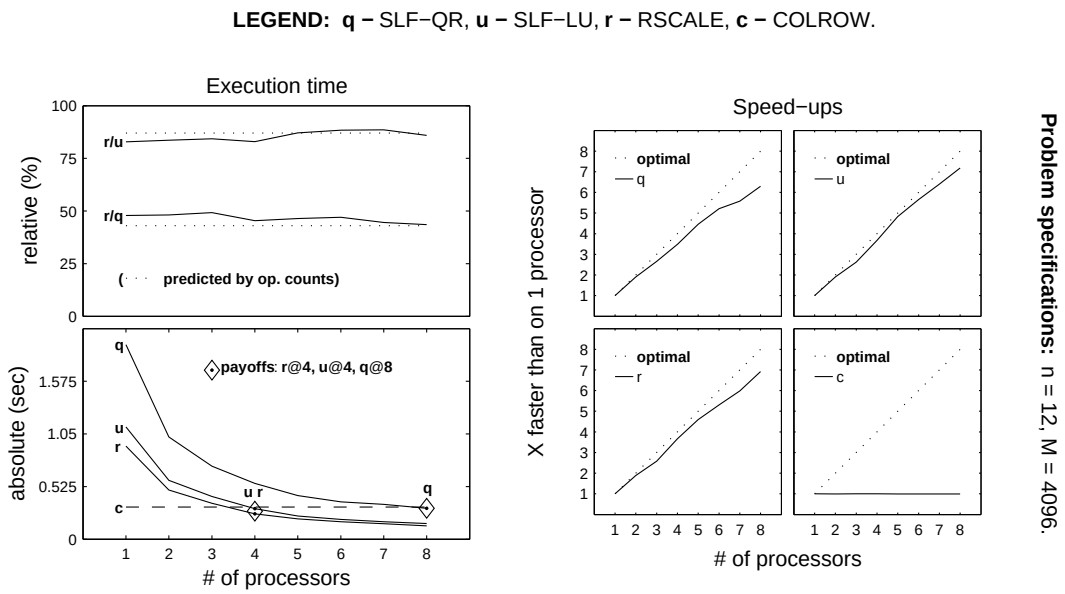


Figure 4.23: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem E of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE, *SLF*-LU and *SLF*-QR payoff at 3, 4 and 7 processors, respectively.

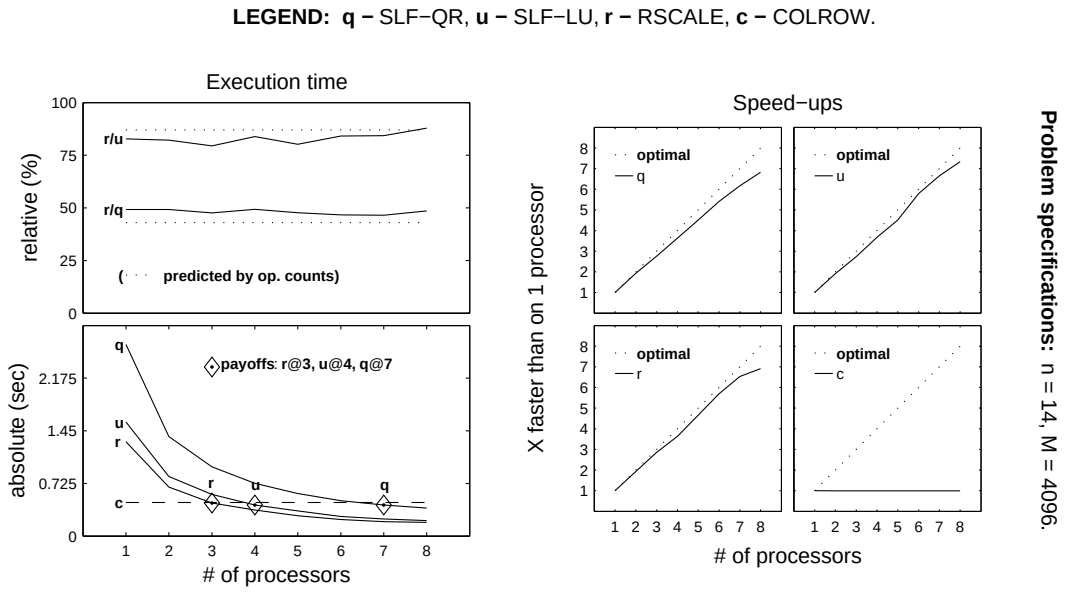
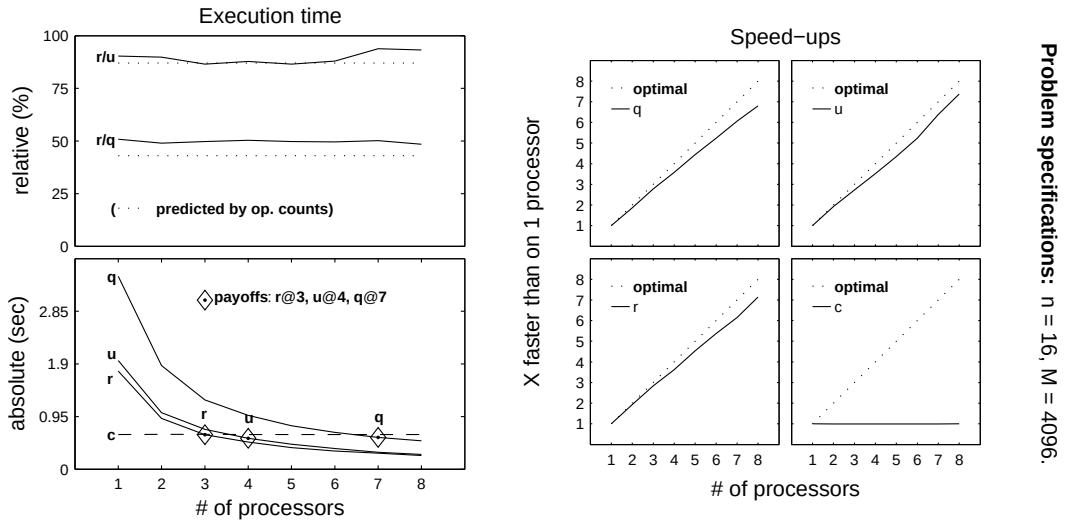


Figure 4.24: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem F of Table 4.5, with $M = 4096$. Architecture is the SGI Origin 2000. RSCALE, *SLF*-LU and *SLF*-QR payoff at 3, 4 and 7 processors, respectively.



4.4 Performance within MirkDC

We now assess the relative performance of the three algorithms—in terms of both accuracy and speed—when the codes are incorporated in MirkDC [Enri 96], a software package for solving nonlinear boundary value ordinary differential equations (BVODEs).

The MirkDC package is designed to solve a system of first-order nonlinear BVODEs, $y'(t) = f(t, y(t))$, with separated boundary conditions. Starting with an initial mesh of M_0 subintervals which partitions the problem interval $[t_a, t_b]$, and an initial guess for the solution, MirkDC proceeds to discretize the continuous problem using one of several mono-implicit Runge-Kutta (MIRK) schemes. The resulting system of nonlinear algebraic equations, or *residual*, is solved using a hybrid damped Newton and fixed Jacobian iteration. As described in §1.1, each Newton iteration involves the evaluation of the residual, (possibly) the construction and factorization an almost block diagonal (ABD) system of linear algebraic equations, and the backsolve of the factored system. Once the Newton iteration has converged, a continuous mono-implicit Runge-Kutta scheme is used to interpolate the computed discrete solution with a $\mathcal{C}^1$ continuous polynomial, $u(t)$, which in turn is used to compute an estimate of the solution defect $\|u'(t) - f(t, u(t))\|$. The final convergence test compares this defect estimate to a user defined tolerance, τ_{defect}. If this test fails, MirkDC uses the defect estimates on each subinterval of the current mesh to design a new mesh which equidistributes the defect estimates. A new initial guess for the solution on the new mesh is computed with $u(t)$, and the process repeats. See [Enri 96] for a more detailed description of the code.

The primary computational costs associated with MirkDC may be attributed to the program segments responsible for the following five tasks:

1. residual evaluation
2. defect estimation
3. ABD matrix construction
4. ABD matrix factorization
5. ABD system backsolve

Other costs, such as those attributable to mesh selection or redistribution, are less significant ([Muir 91], [Muir 03]). In the following sections we assess the overall and task-specific performance of four variants of MirkDC:

- MirkDC/COLROW
- MirkDC/*SLF*-QR
- MirkDC/*SLF*-LU
- MirkDC/RSCALE

As indicated by their names, these variants differ only in how the tasks of ABD matrix factorization and system backsolve are handled. In MirkDC/COLROW, each ABD system solution is computed with COLROW [Diaz 83]; in MirkDC/*SLF*-QR, MirkDC/*SLF*-LU and MirkDC/RSCALE, each ABD system solution is computed with the *SLF*-QR, *SLF*-LU and RSCALE parallel solvers, respectively. All MirkDC experiments below involve the numerical solution of the nonlinear problem *Swirling Flow III* (SWF-III) [Asch 88], which, when transformed into a first-order system, may be written as

$$\begin{bmatrix} y_1' \\ y_2' \\ y_3' \\ y_4' \\ y_5' \\ y_6' \end{bmatrix} = \begin{bmatrix} y_2 \\ (1/\epsilon)(y_1 y_4 - y_3 y_2) \\ y_4 \\ y_5 \\ y_6 \\ -(1/\epsilon)(y_3 y_6 + y_1 y_2) \end{bmatrix} \tag{4.16}$$

subject to the boundary conditions

$$y_1(t_a) = -1, \quad y_1(t_b) = 1, \quad y_3(t_a) = y_4(t_a) = 0, \quad y_3(t_b) = y_4(t_b) = 0. \tag{4.17}$$

These equations model the "swirling" flow of a viscous incompressible fluid between two counter-rotating coaxial disks located at $t = t_a$ and $t = t_b$. The degree of viscosity is specified by $\epsilon, 0 < \epsilon \leq 1$. The problem is more difficult numerically for less viscous fluids; i.e. for small values of ϵ which correspond to large Reynolds numbers. No closed form solution exists.

Experiments are run on three different computer architectures, as specified in Table 4.6. Of particular interest in the experiments is the *convergence pattern* of each MirkDC variant. This concept will be defined formerly below; simply stated it refers to the sequence of meshes chosen by MirkDC as the computation proceeds, and the number of full Newton iterations associated with each of the meshes. In §4.4.1 and §4.4.2, we assess the sequential and parallel performance, respectively, of the four variants of MirkDC using experiments in which the convergence patterns (1) are *identical* among variants, and (2) are not affected by partitioning.

Table 4.6: Architecture specifications. The Sun Ultra 2 is used as a sequential machine.

Architecture			Specificiations			
			processors			memory
acronym	model	vendor	#	speed	type	
CHA	Challenge L	Silcon Graphics	8	150 MHZ	R4400	512 Megabytes
ORG	Origin 2000	Silcon Graphics	8	250 MHZ	R10000	2 Gigabytes
ULT	Ultra 2/2170	Sun Microsystems	2	150 MHZ	SPARC	256 Megabytes

This second point is especially important when illustrating speedups in §4.4.2. In §4.4.3 and §4.4.4, we assess the sequential performance of the four variants of MirkDC using experiments in which the convergence patterns *differ* among variants and/or are affected by partitioning. In §4.4.3, we analyze problems where an unusual convergence pattern in MirkDC/*SLF*-LU leads to poor MirkDC/*SLF*-LU performance. We suspect the cause of this performance degradation is *SLF*-LU instability and show that this indeed is the case by extracting selected ABD systems from the MirkDC/*SLF*-LU solution sequence and showing that *SLF*-LU—and only *SLF*-LU—exhibits instability when solving these systems. In §4.4.4 we analyze experiments in which the convergence pattern of MirkDC/RSCALE is shorter than that of the other variants, including that of MirkDC/COLROW. These results suggest that on a sequential machine it sometimes may prove beneficial to choose RSCALE over other state-of-the-art sequential ABD system solvers such as COLROW.

4.4.1 Sequential MirkDC

Each of the experiments discussed in this section was run on the Sun Ultra 2, and involved a sequential single-partition variant of the ABD system solvers. All code was compiled using the level 3 (extensive) optimization options available with Sun's Fortran 77 compiler, and most timing results are averaged over 4-8 runs. The objective of the experiments is to measure the relative performance of the four variants of MirkDC on a sequential machine, and to demonstrate how sequential performance is affected by certain problem and solution strategy parameters. Specifically, we are interested in how the SWF-III problem parameters of

- viscosity ϵ, and
- disk spacing $|t_a - t_b|$,

Table 4.7: Eight MirkDC/SWF-III experiments to measure the relative performance of the four MirkDC variants on a sequential machine and to demonstrate how sequential performance is affected by problem and solution strategy parameters. A SWF-III problem is defined by ϵ and the interval of integration $[t_a, t_b]$. A MirkDC solution strategy is specified by a MIRK scheme, defect tolerance τ_{defect}, number of initial mesh subintervals M_0, and number of partitions.

exp. #	SWF-III parameters		MirkDC solution strategy			
	ϵ	$[t_a, t_b]$	MIRK scheme	τ_{defect}	M_0	# part.
1	0.002	$[0, 1]$	trapezoidal, 2^{nd} order	10^{-5}	10	1
2	0.002	$[0, 1]$	Lobatto, 4^{th} order	10^{-5}	10	1
3	0.002	$[0, 1]$	Lobatto, 4^{th} order	10^{-7}	10	1
4	0.002	$[0, 1]$	6^{th} order	10^{-7}	10	1
5	0.002	$[-1, 1]$	trapezoidal, 2^{nd} order	10^{-5}	10	1
6	0.002	$[-1, 1]$	Lobatto, 4^{th} order	10^{-5}	10	1
7	0.002	$[-1, 1]$	Lobatto, 4^{th} order	10^{-7}	10	1
8	0.002	$[-1, 1]$	6^{th} order	10^{-7}	10	1

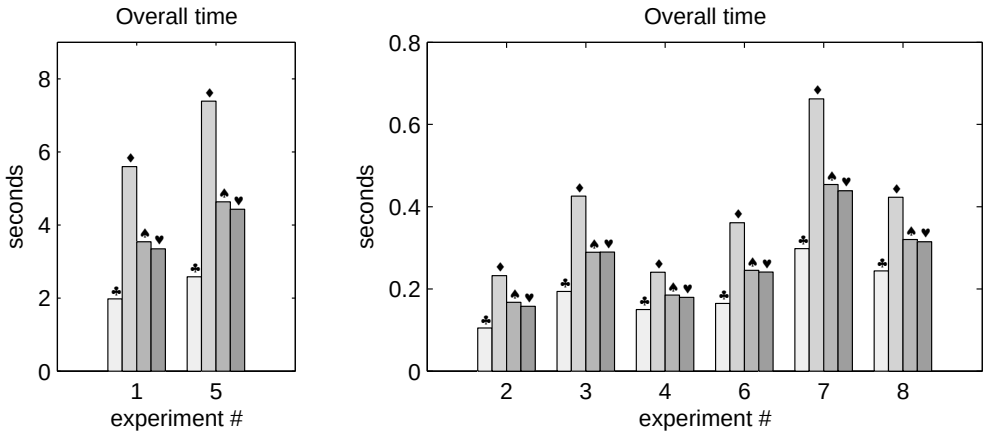
and MirkDC solution strategy parameters of

- MIRK discretization scheme order,
- defect tolerance τ_{defect},
- number of initial mesh subintervals M_0, and
- number of partitions

affect MirkDC performance.

Table 4.7 lists eight experiments in which the SWF-III problems solved are not difficult numerically—at least not compared to other experiments presented later in this section. These experiments demonstrate some effects of disk spacing, MIRK scheme order and defect tolerance on MirkDC performance. Figure 4.25 shows the absolute overall execution time of the four variants of MirkDC in each experiment. Note that the execution times of experiments #1 and #5, shown separately in the bar graph on the left, are at least an order of magnitude greater

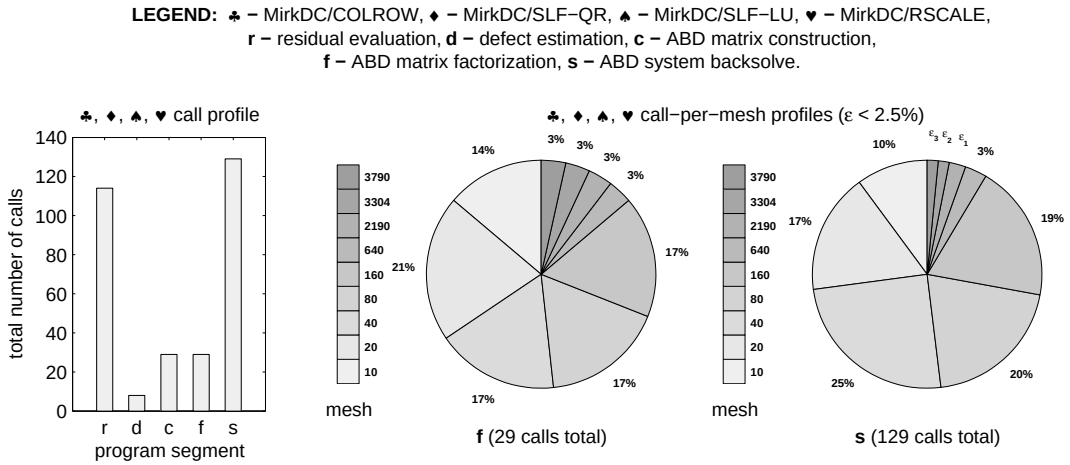
Figure 4.25: Overall execution time of the four variants of MirkDC when run on the experiments specified in Table 4.7. The relative performance of the variants reflects that of the underlying ABD system solvers. Note the execution times of experiments #1 and #5 are an order of magnitude greater than those of the other experiments.



than those of the other six experiments. This illustrates a consequence of choosing an inadequate discretization scheme (2^{nd} order) for the required defect tolerance $\tau_{\text{defect}} = 10^{-5}$. The graphs for experiments #(3,4) and #(7,8) illustrate this as well, but to a lesser extent. Comparing the graphs for experiments #(2,3) and #(6,7), we see an effect of decreasing the defect tolerance. Comparing the graphs for experiments #(1,5), #(2,6), #(3,7) and #(4,8), we see an effect of increasing the disk spacing.

Figure 4.25 also shows that the *relative* performance of the four variants of MirkDC in each experiment is consistent with what one would expect given the relative performance of the underlying ABD system solvers (§4.1 and §4.2); i.e. ranking from fastest to slowest we have (1) MirkDC/COLROW, (2) MirkDC/RSCALE, (3) MirkDC/*SLF*-LU and (4) MirkDC/*SLF*-QR. This is true in part because, in each experiment, the MirkDC variants exhibit identical *convergence patterns*. We define the convergence pattern of a MirkDC experiment in terms of its subroutine call and call-per-mesh profile. The subroutine call profile summarizes the total number of calls to the program segments responsible for residual evaluation, defect estimation, ABD matrix construction, ABD matrix factorization and ABD system backsolve; i.e. the five tasks comprising the primary computational costs associated with MirkDC. The call-per-mesh profile summarizes the number of ABD matrix factorizations and ABD system backsolves performed on each mesh selected by MirkDC as the computation proceeds, and thus also discloses

Figure 4.26: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #1 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

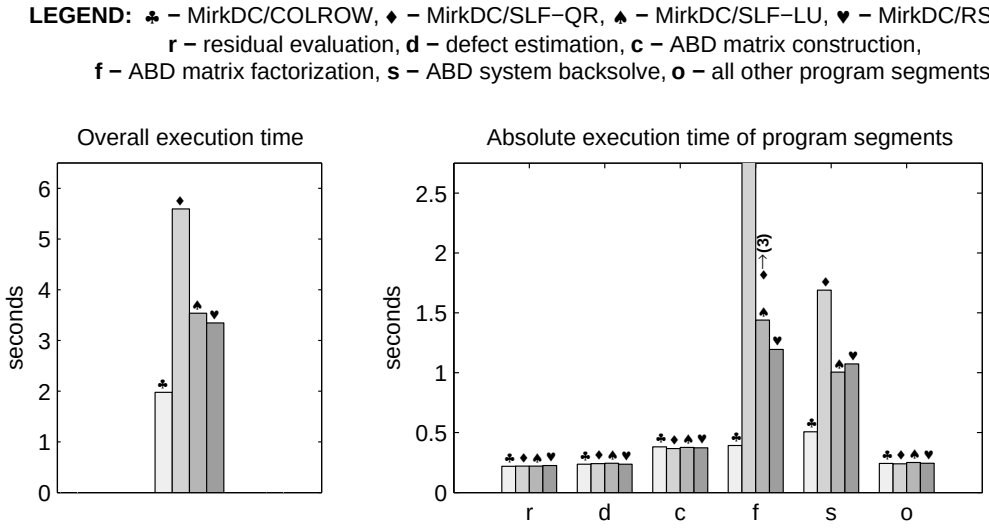


the mesh selection strategy.

Figure 4.26 shows the convergence pattern of the four variants of MirkDC in experiment #1 of Table 4.7. From the call-per-mesh profiles, we see that MirkDC selects meshes of size 10 to 3790 subintervals. The initial uniform mesh of $M_0 = 10$ subintervals is specified in the solution strategy; subsequent meshes are, in general, non-uniform. The pie charts give the number of factorizations and backsolves on each mesh. For example, on the mesh of size 160 there were 17% of 29 or 5 factorizations and 19% of 129 or 25 backsolves. (The percentages in the pie charts are rounded to two figures.) From the call profile we see that, as is always the case, there are exactly the same number of ABD matrix constructions and factorizations. There are relatively few defect estimations, as these are computed during the the final convergence test only. Finally, the number of residual evaluations is close to—but not exactly the same as—the number of ABD system backsolves. See [Enri 96] for an explanation of why these totals sometimes differ.

Figure 4.27 shows a breakdown of the execution times of the five tasks profiled in Figure 4.26. Also shown is the cumulative execution time of all other program segments. From this figure, it is clear that the differences in the overall execution time of the four variants of MirkDC is attributable solely to the differences in factorization and backsolve times of the four underlying ABD system solvers. This is always the case when the MirkDC variants exhibit identical convergence patterns. In §4.4.3 and §4.4.4, we consider experiments where the overall execution time of the variants differs for other reasons.

Figure 4.27: Overall and selected program segment execution times of the four variants of MirkDC in experiment #1 of Table 4.7. The differences in overall execution time are attributable solely to the differences in factorization and backsolve times of the four underlying ABD system solvers.



Figures 4.28 and 4.29 show the task execution times and convergence patterns, respectively, in experiment #2 of Table 4.7. Comparing Figures 4.28 and 4.27, we see that switching to a 4^{th} order MIRK scheme in experiment #2 results in a substantial reduction in all task execution times. Comparing Figures 4.29 and 4.26, it is clear that this reduction in execution times is due in part to fewer subroutine calls overall, but primarily can be attributed to working with smaller ABD systems built on meshes of fewer subintervals—the largest ABD system arising in experiment #2 consists of only 103 block-rows. Note also when comparing Figures 4.28 and 4.27, that the ratio of ABD matrix construction to factorization time *increases* in experiment #2. (Recall that there is always the same number of ABD matrix constructions and factorizations in a given experiment.) This increase is due to the higher computational cost associated with a 4^{th} order discretization.

Figures 4.30 and 4.31 show the task execution times and convergence patterns, respectively, in experiment #3 of Table 4.7. Comparing Figures 4.30 and 4.28, we see that switching to a stricter defect tolerance in experiment #3 results in a moderate increase in all task execution times. Comparing Figures 4.31 and 4.29, it is clear that this increase in execution times primarily can be attributed to working with moderately larger ABD systems.

Finally, Figures 4.32 and 4.33 show the task execution times and convergence patterns,

Figure 4.28: Overall and selected program segment execution times of the four variants of MirkDC in experiment #2 of Table 4.7. Switching to a 4^{th} order MIRK scheme results in a substantial reduction in execution times. Compare to Figure 4.27, and compare the profiles in Figures 4.26 and 4.29.

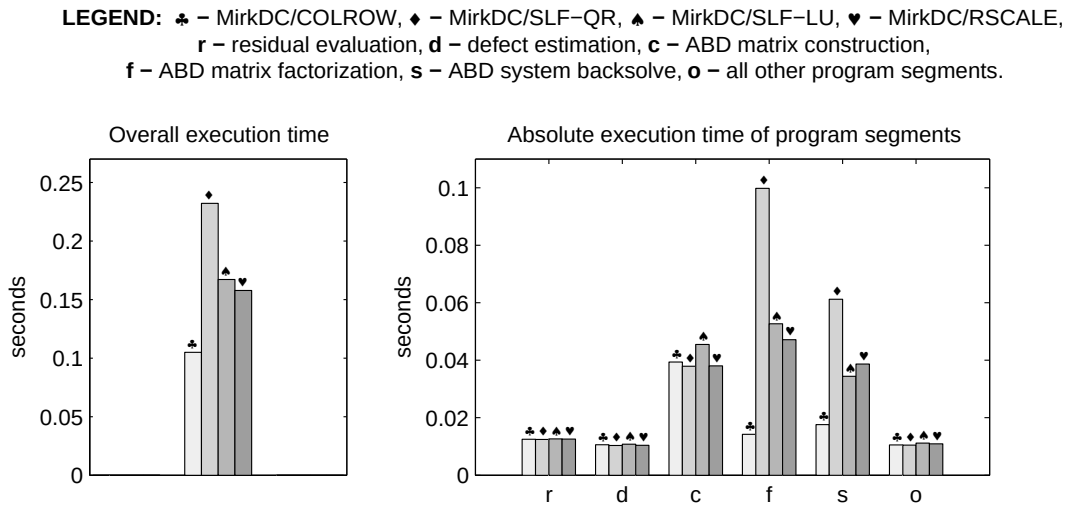


Figure 4.29: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #2 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

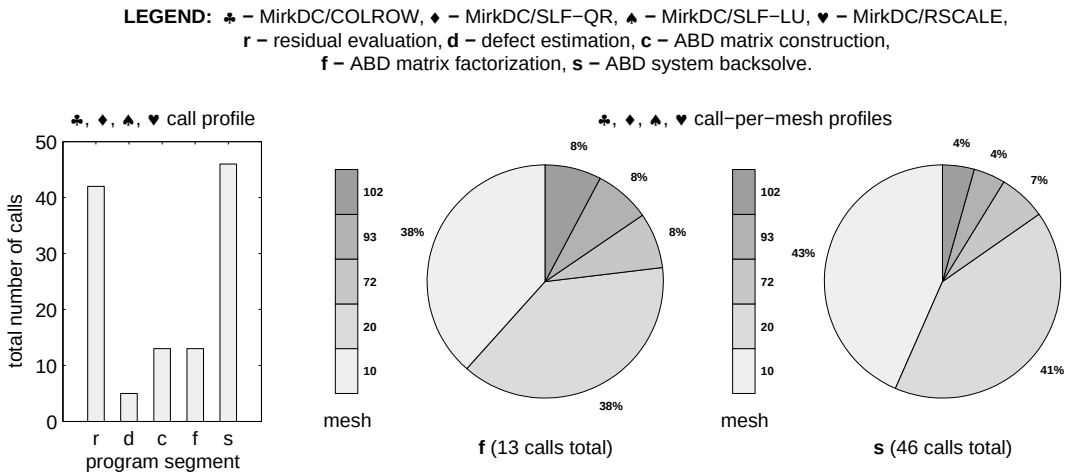


Figure 4.30: Overall and selected program segment execution times of the four variants of MirkDC in experiment #3 of Table 4.7. Switching to a stricter defect tolerance results in a moderate increase in execution times. Compare to Figure 4.28, and compare the profiles in Figures 4.29 and 4.31.

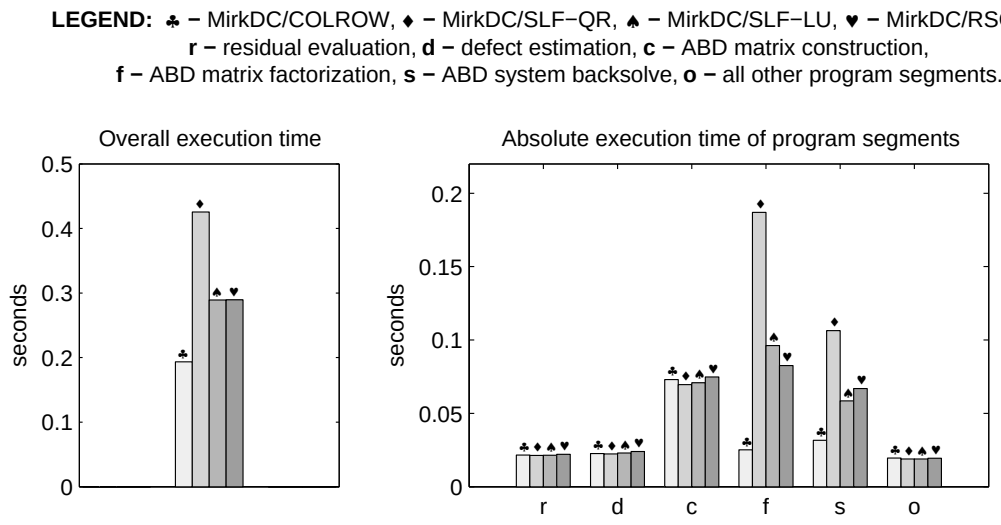


Figure 4.31: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #3 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

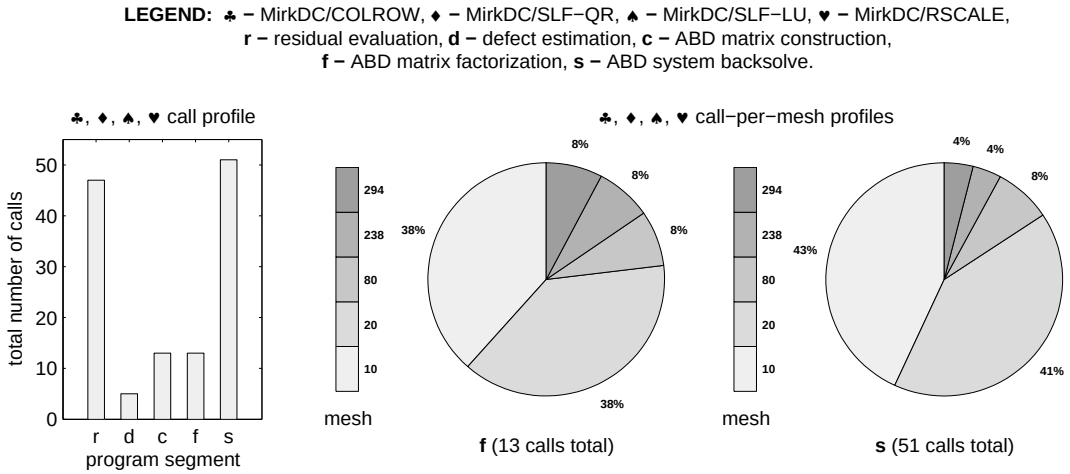


Figure 4.32: Overall and selected program segment execution times of the four variants of MirkDC in experiment #4 of Table 4.7. Switching to a 6^{th} order MIRK scheme results in a moderate reduction in execution times. Compare to Figure 4.30, and compare the profiles in Figures 4.31 and 4.33.

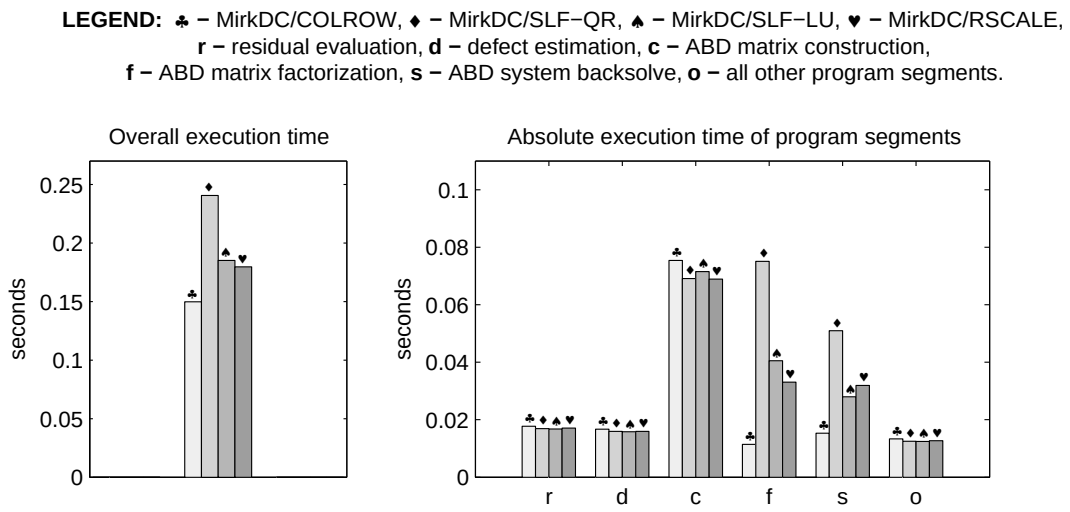


Figure 4.33: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #4 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

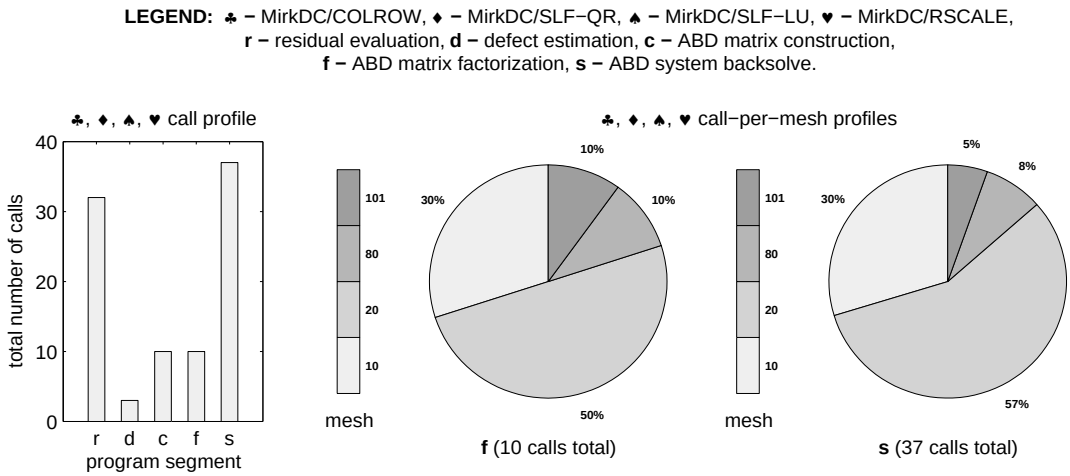


Table 4.8: Eight MirkDC/SWF-III experiments to measure the relative performance of the four MirkDC variants on a sequential machine and to demonstrate how sequential performance is affected by problem and solution strategy parameters. The problems in these experiments are difficult numerically and must be solved using parameter continuation. A SWF-III problem is defined by ϵ and the interval of integration $[t_a, t_b]$. A MirkDC solution strategy is specified by a MIRK scheme, defect tolerance τ_{defect}, number of initial mesh subintervals M_0, number of partitions, and parameter continuation strategy.

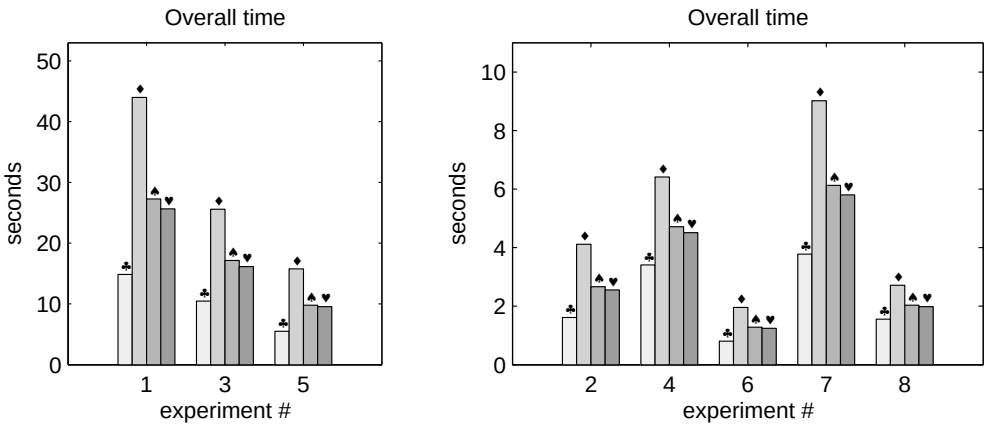
exp. #	SWF-III parameters		MirkDC solution strategy				
	ϵ	$[t_a, t_b]$	scheme	τ_{defect}	M_0	# part.	ϵ continuation
1	0.0015	$[0, 2]$	2^{nd} ord.	10^{-5}	20	1	$\{10, 4, 1.5\} \times 10^{-3}$
2	0.0015	$[0, 2]$	4^{th} ord.	10^{-7}	20	1	$\{10, 4, 1.5\} \times 10^{-3}$
3	0.0015	$[0, 2]$	4^{th} ord.	10^{-10}	20	1	$\{10, 4, 1.5\} \times 10^{-3}$
4	0.0015	$[0, 2]$	6^{th} ord.	10^{-10}	20	1	$\{10, 4, 1.5\} \times 10^{-3}$
5	0.0015	$[-1, 3]$	2^{nd} ord.	10^{-5}	20	1	$\{100, 50, 1.5\} \times 10^{-3}$
6	0.0015	$[-1, 3]$	4^{th} ord.	10^{-7}	20	1	$\{100, 50, 1.5\} \times 10^{-3}$
7	0.0015	$[-1, 3]$	4^{th} ord.	10^{-10}	20	1	$\{100, 50, 1.5\} \times 10^{-3}$
8	0.0015	$[-1, 3]$	6^{th} ord.	10^{-10}	20	1	$\{100, 50, 1.5\} \times 10^{-3}$

respectively, in experiment #4 of Table 4.7. Here we see many of the same effects as in experiment #2; namely that switching to a 6^{th} order MIRK scheme in experiment #4 results in a moderate reduction in all task execution times, and the ratio of ABD matrix construction to factorization time increases over that in experiment #3. Figures summarizing the task execution times and convergence patterns in experiments #5-#8 are included in Appendix D.1.

While we are able to demonstrate some sequential performance characteristics of the four MirkDC variants with the experiments listed in Table 4.7, these experiments are not suitable for measuring parallel performance because they are not computationally intensive; the overall execution time in most is less than a second. Thus, in preparation for the discussion of parallel performance in the next section, we present eight additional MirkDC/SWF-III experiments in Table 4.8. Each of these new experiments is more computationally intensive than any of those listed in Table 4.7, for one or more of the following reasons:

- ϵ is smaller,

Figure 4.34: Overall execution time of the four variants of MirkDC when run on the experiments specified in Table 4.8. The relative performance of the variants reflects that of the underlying ABD system solvers. Note the execution times of experiments #1, #3 and #5 are displayed on a different time scale than those of the other experiments.

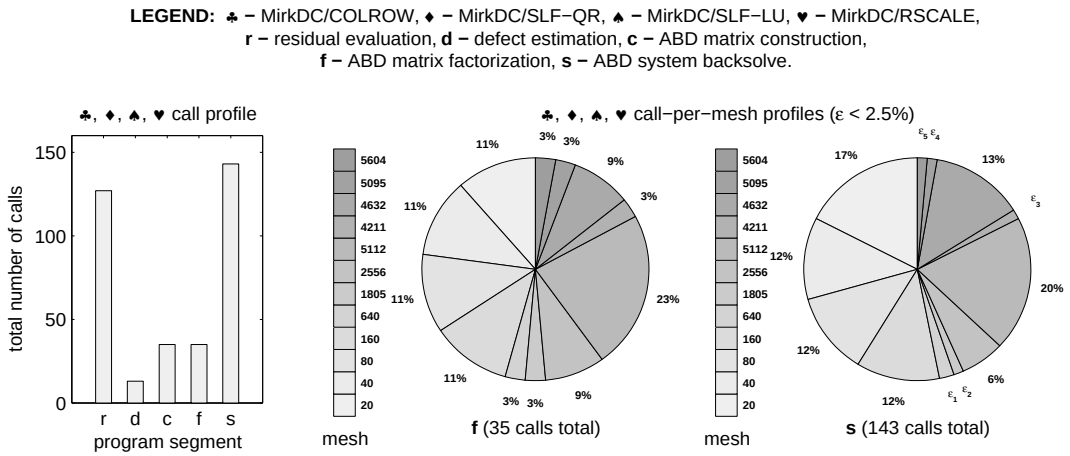


- the disk spacing is wider,
- the defect tolerance is stricter.

Figure 4.34 shows the absolute overall execution time of the four variants of MirkDC in each experiment listed in Table 4.8. As in Figure 4.25, we see that the relative performance of the four variants of MirkDC is consistent with what one would expect given the relative performance of the underlying ABD system solvers. Also, as desired, the experiments in Table 4.8 are clearly more computationally intensive than those in Table 4.7.

Designing more computationally intensive MirkDC experiments is not as straightforward as it may seem. Adjusting problem and/or solution strategy parameters as suggested above does increase execution time to a point, however before times increase sufficiently MirkDC often begins to exhibit converge difficulties. When this happens, parameter continuation can be used to help MirkDC converge. Each of the experiments listed in Table 4.8 incorporates a form of parameter continuation—involving ϵ—as part of its solution strategy; the continuation sequence is specified in the column titled "ϵ continuation". For example, in experiment #1 we begin with a uniform mesh of $M_0 = 20$ subintervals and then proceed to solve the specified SWF-III problem ($\epsilon = 0.0015$) using three continuation iterations. During the first iteration, we solve a comparatively easy SWF-III problem with $\epsilon = 0.01$. We then use the solution and

Figure 4.35: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #1 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 2556, 4632 and 5604 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.



final mesh from the first iteration as the initial solution and mesh for a second SWF-III problem with $\epsilon = 0.004$. Finally, we use the solution and final mesh from the second iteration as the initial solution and mesh for a third SWF-III problem with $\epsilon = 0.0015$.

Figure 4.35 shows the convergence pattern of the four variants of MirkDC in experiment #1 of Table 4.8. The profiles shown represent subroutine calls counted over all three continuation iterations. The first, second and third continuation iteration terminates with a final mesh of 2556, 4632 and 5604 subintervals, respectively. (This information is extracted from a more detailed profile not shown in the figure.) Figure 4.36 shows a breakdown of the execution times of the five tasks profiled in Figure 4.35. These times are accumulated over all three continuation iterations. Again, this figure shows that the differences in the overall execution time of the four variants of MirkDC is attributable solely to the differences in factorization and backsolve times of the four underlying ABD system solvers.

It should be noted that parameter continuation is not just an option in the experiments listed in Table 4.8; it is necessary for convergence. For example, if we try to solve the specified SWF-III problem in experiment #1 using MirkDC/COLROW *without* continuation, the convergence pattern shown in Figure 4.37 arises. MirkDC/COLROW no longer converges in this experiment; instead it proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals (12000 in this implementation) is

Figure 4.36: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #1 of Table 4.8. The differences in overall execution time are attributable solely to the differences in factorization and backsolve times of the four underlying ABD system solvers.

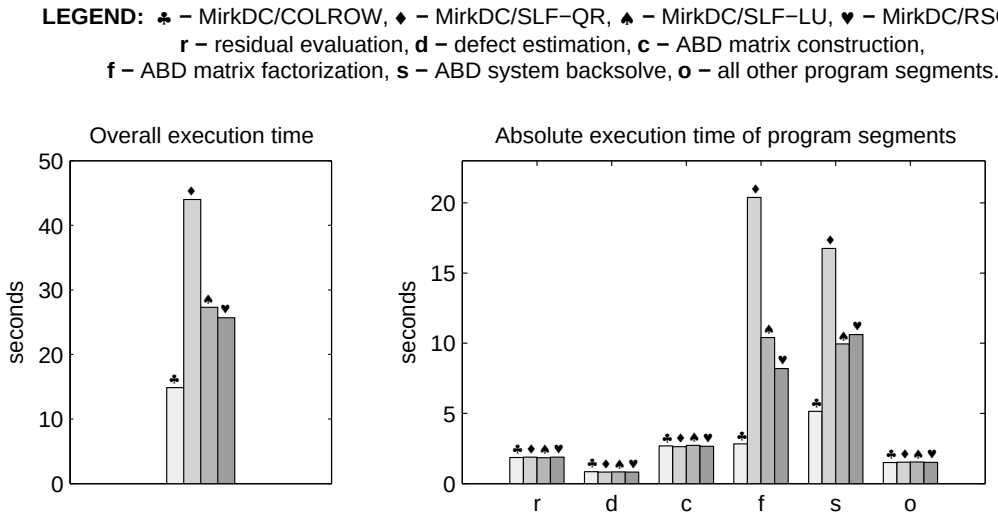


Figure 4.37: Subroutine call and call-per-mesh profiles of MirkDC/COLROW in experiment #1 of Table 4.8, *when parameter continuation is not used*. Without continuation, MirkDC/COLROW does not converge. The code proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals is exceeded; the defect estimation stage is never reached.

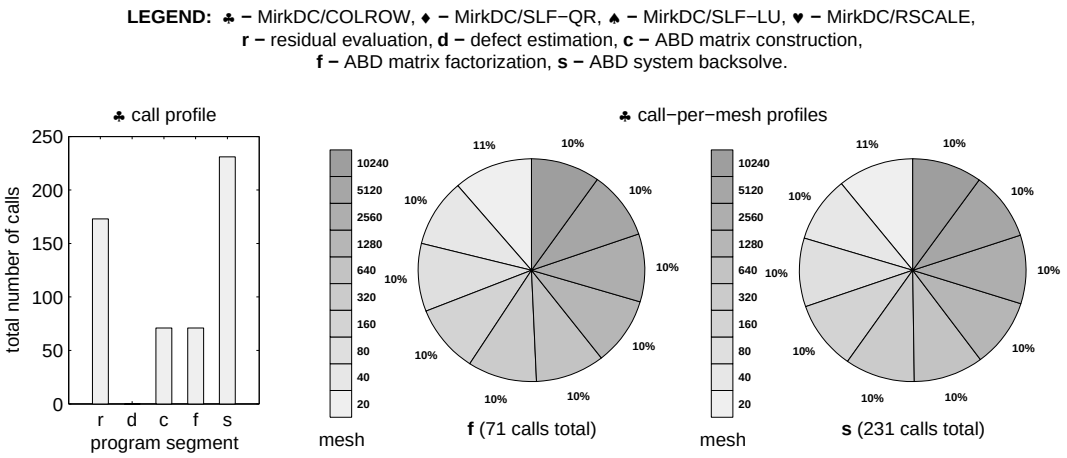
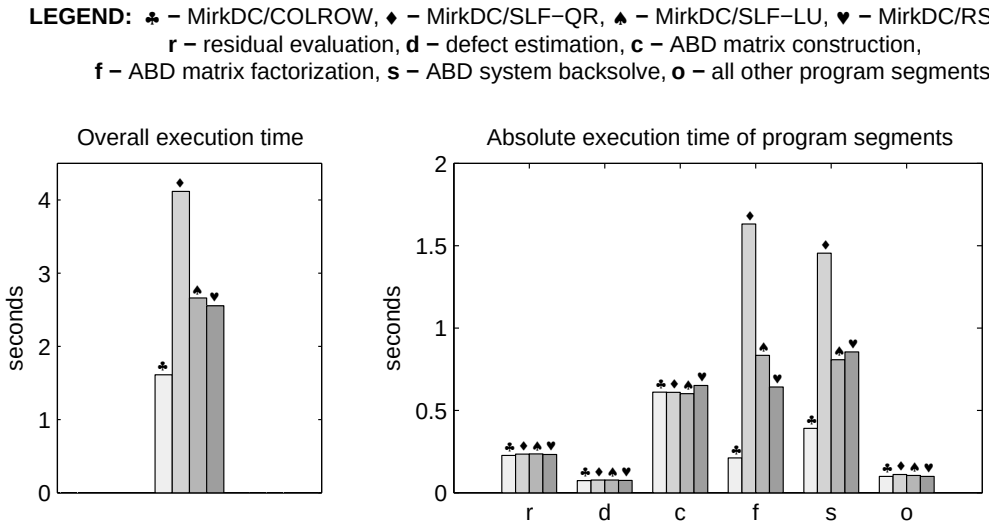


Figure 4.38: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #2 of Table 4.8. Switching to a 4^{th} order MIRK scheme results in a substantial reduction in execution times, even with the stricter defect tolerance. Compare to Figure 4.36, and compare the profiles in Figures 4.35 and 4.39.



exceeded. As indicated by the 0 count in the subroutine call profile, the defect estimation stage is never reached. Similar behaviour is observed in each experiment in Table 4.8 when parameter continuation is not used.

Figures 4.38, 4.40, 4.42 and 4.39, 4.41, 4.43 show the task execution times and convergence patterns, respectively, in experiments #2, #3, #4 of Table 4.8. These experiments demonstrate many of the same performance effects as the experiments in Table 4.7, but with greater execution time. Of particular interest is experiment #4, where the third continuation iteration terminates with a mesh of *fewer* subintervals than the second. In this experiment, MirkDC designed a new mesh with fewer subintervals than the preceding mesh when equidistributing the defect estimate during the third continuation iteration. This is not so uncommon in difficult problems—the number of subintervals does not always strictly increases in the mesh sequence of a MirkDC convergence pattern. Figures summarizing the task execution times and convergence patterns in experiments #5-#8 are included in Appendix D.1. Note that the wider disk spacing in experiments #5-#8 does not necessarily lead to greater execution time since a different continuation sequence must be used in order to achieve convergence.

Figure 4.39: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #2 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 244, 349 and 399 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

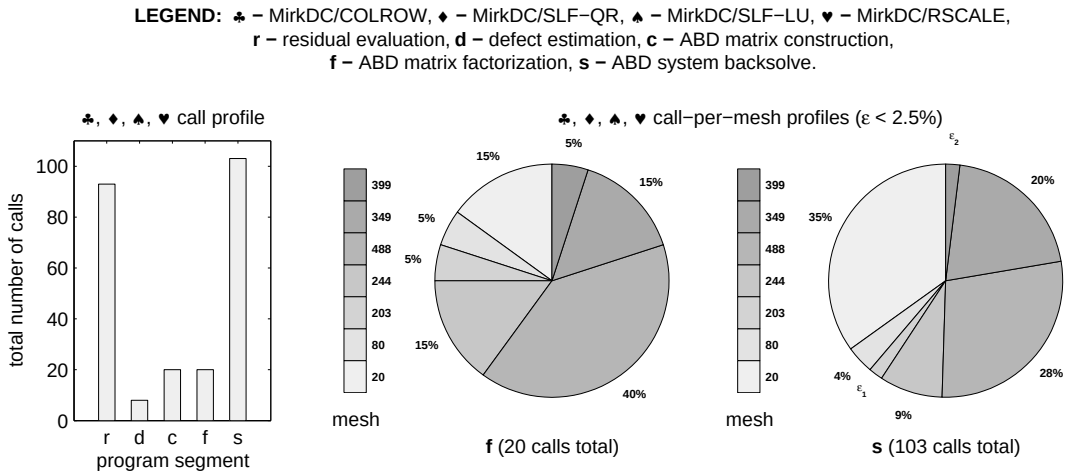


Figure 4.40: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #3 of Table 4.8. Switching to a stricter defect tolerance results in a moderate increase in execution times. Compare to Figure 4.38, and compare the profiles in Figures 4.39 and 4.41.

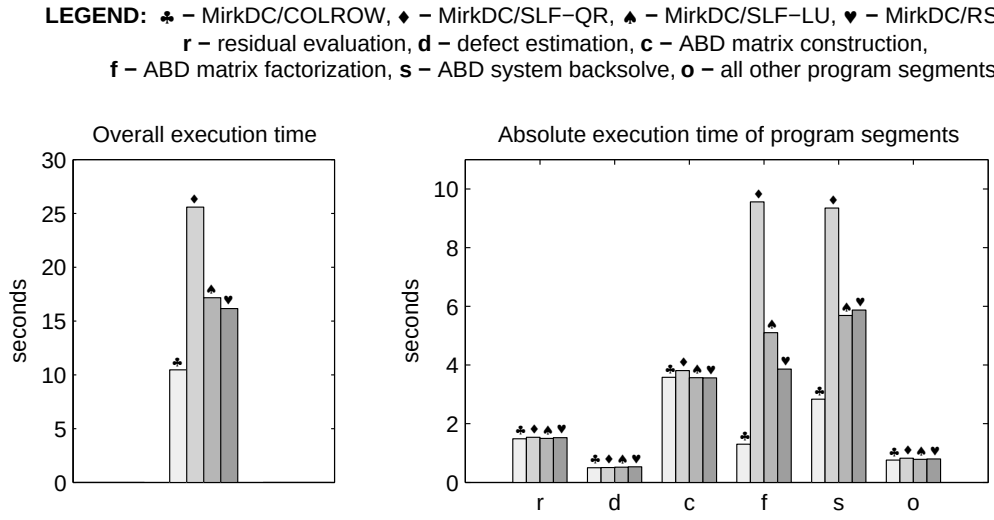


Figure 4.41: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #3 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 1350, 1947 and 2448 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

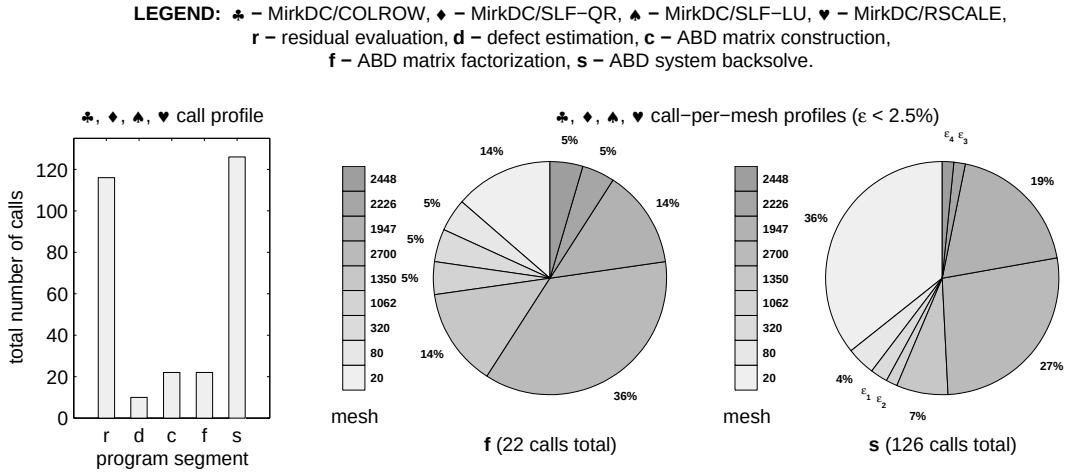


Figure 4.42: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #4 of Table 4.8. Switching to a 6^{th} order MIRK scheme results in a moderate reduction in execution times. Compare to Figure 4.40, and compare the profiles in Figures 4.41 and 4.43.

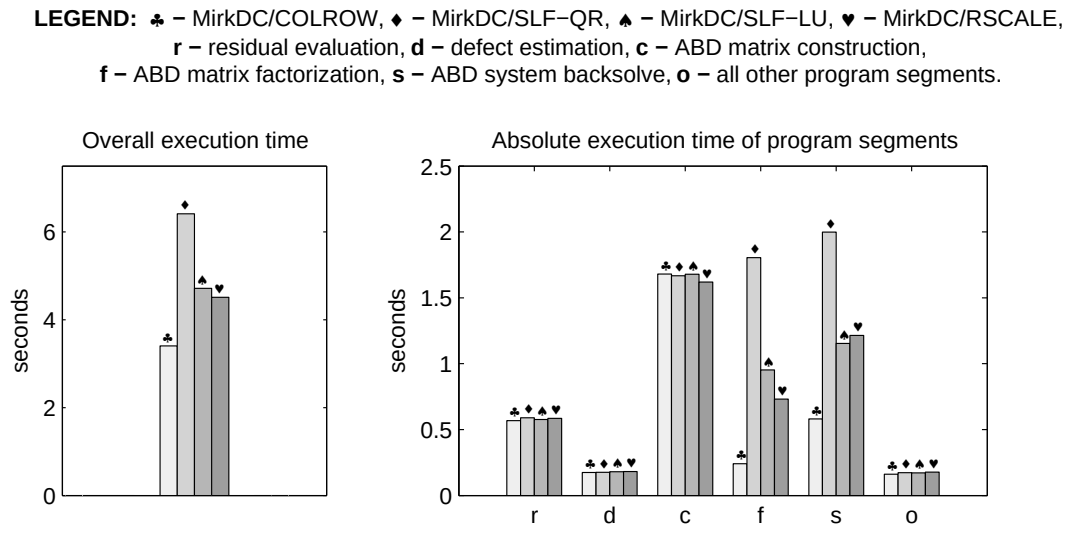
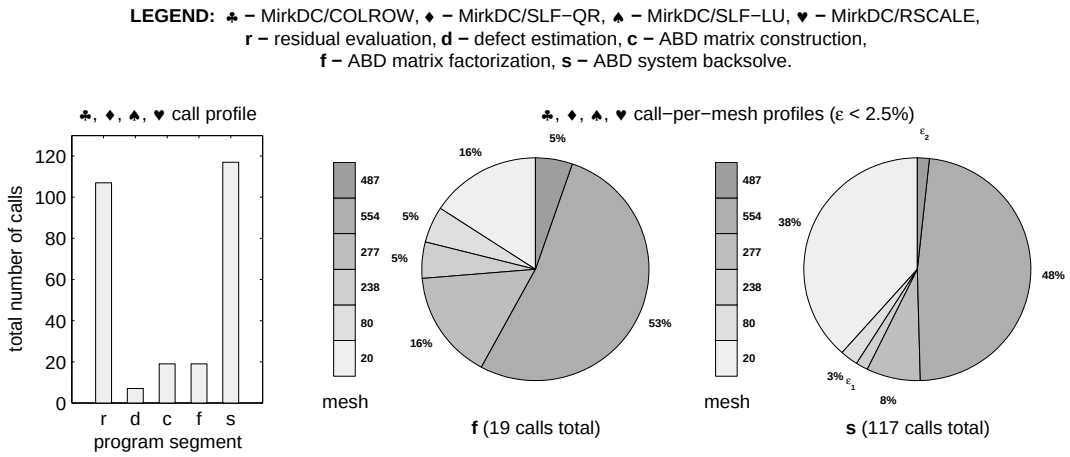


Figure 4.43: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #4 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 277, 554 and 487 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.



4.4.2 Parallel MirkDC

The experiments in the previous section—especially those in Table 4.8—clearly show that ABD matrix construction, factorization and backsolve dominate the computational costs associated with MirkDC. The costs associated with residual evaluation, defect estimation, and the sum total of all other program segments are not negligible, but they are far less significant. Hence, the greatest gain from parallelism in MirkDC should be realized by parallelizing the three tasks associated with the ABD matrix.

For the experiments in this section, we have developed and tested code on two parallel computer architectures: the Challenge L and Origin 2000. Both are a tightly-coupled, shared-memory machines; the latter is faster and has more memory. Complete architecture specifications are given in Table 4.6. Parallelism and memory partitioning on both machines is achieved by using shared-memory parallel compiler directives provided by the Silicon Graphics Fortran 77 compiler. The MirkDC tasks of ABD matrix construction, residual evaluation and defect estimation may be parallelized in a straightforward manner with these directives, using the loop partitioning strategy discussed in §4.3. The parallelization of ABD matrix factorization and backsolve is, of course, the main goal of our work in this thesis, and in theory could be handled by any one of the parallel ABD system solvers. Since we have developed three par-

Table 4.9: Five SWF-III problems. Each problem is specified by ϵ and the left and right endpoints of the interval of integration (i.e. the distance between the rotating disks). The problems are listed in increasing order of difficulty.

Problem #	SWF-III parameters	
	ϵ	$[t_a, t_b]$
A	.002	$[0, 1]$
B	.000125	$[0, 1]$
C	.000125	$[-1, 1]$
D	.0001	$[-1, 1]$
E	.00275	$[0, 10]$

allel ABD system solvers, we have three possibilities for a parallel variant of MirkDC. We discuss and present numerical results for only one of these variants in this section—parallel MirkDC/RSCALE.

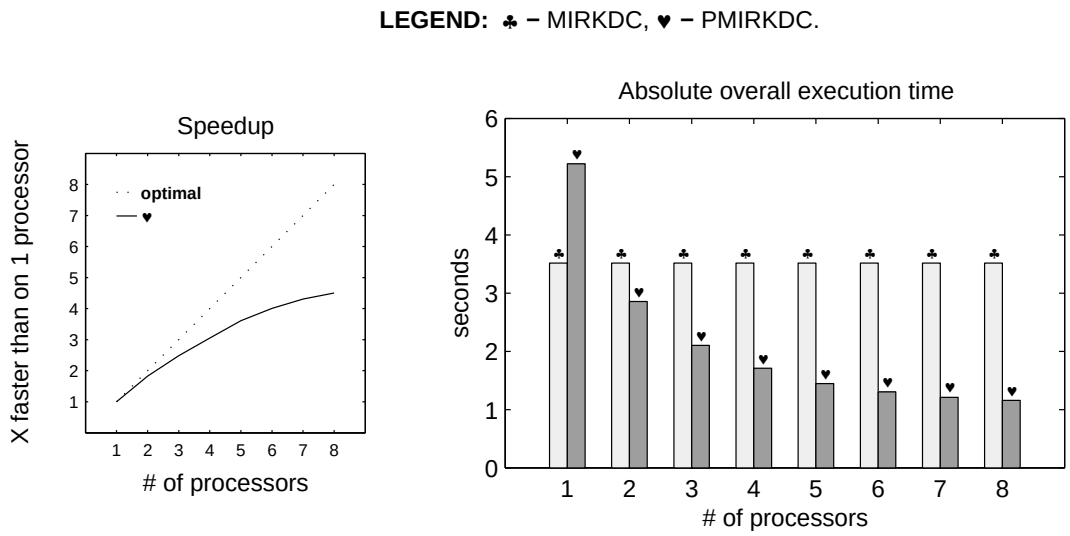
Parallel MirkDC/RSCALE has already appeared in the literature. It was first published as a technical report, and then appeared in Parallel Computing [Muir 03]. In this paper, the accuracy and speed of parallel MirkDC/RSCALE is compared to that of sequential MirkDC/COLROW. The experiments and numerical results discussed in [Muir 03] were prepared during the writing of this thesis. We review these results in this section. In order to be consistent with the nomenclature used in the paper, throughout this section we refer to sequential MirkDC/COLROW as simply MirkDC, and parallel MirkDC/RSCALE as PMirkDC.

Table 4.9 specifies parameters for five SWF-III problems. The problems are listed in increasing order of difficulty. A problem becomes more difficult as the magnitude of ϵ decreases and/or the length of the interval of integration increases. Table 4.10 lists specifications for ten experiments used to compare the performance of MirkDC and PMirkDC. A MirkDC solution strategy is as defined in §4.4.1. Experiments are actually run on *three* different architectures: the two parallel machines mentioned above and the sequential Sun Ultra 2. We use a sequential machine for experiments #9 and #10 because the convergence patterns of PMirkDC and MirkDC differ significantly in these experiments. As the convergence pattern has a major effect on performance, it does not make sense to test on a parallel machine. In other words, speedup is not the issue in these experiments. We discuss this further at the end of the section.

Table 4.10: Ten MirkDC vs. PMirkDC experiments. Each experiment is specified by a host architecture (Table 4.6), a SWF-III problem (Table 4.9), and a MirkDC solution strategy. A MirkDC solution strategy is given by a MIRK scheme, defect tolerance τ_{defect}, number of initial mesh subintervals M_0, and parameter continuation strategy.

exp. #	arch.	prb.	MirkDC solution strategy			
			scheme	τ_{defect}	M_0	ϵ continuation
1	CHA	A	4^{th} ord.	10^{-11}	7000	none; solve directly for $\epsilon = .002$
2	CHA	A	4^{th} ord.	10^{-11}	10	none; solve directly for $\epsilon = .002$
3	ORG	A	4^{th} ord.	10^{-11}	7000	none; solve directly for $\epsilon = .002$
4	ORG	A	4^{th} ord.	10^{-11}	10	none; solve directly for $\epsilon = .002$
5	ORG	B	4^{th} ord.	10^{-8}	10	$\{.002, .001, .0005, .00025, .000125\}$
6	ORG	C	4^{th} ord.	10^{-6}	10	$\{.002, .001, .0005, .00025, .000125\}$
7	ORG	D	4^{th} ord.	10^{-7}	10	$\{.002, .001, .0004, .0002, .0001\}$
8	ORG	E	4^{th} ord.	10^{-7}	10	$\{1, .1, .01, .005, .00275\}$
9	ULT	C	4^{th} ord.	10^{-6}	10	none; solve directly for $\epsilon = .000125$
10	ULT	D	4^{th} ord.	10^{-7}	10	none; solve directly for $\epsilon = .0001$

Figure 4.44: Overall speedup and execution time of MirkDC and PMirkDC in experiment #2 of Table 4.10. Parallelism begins to pay-off at 2 processors.



Problem A is not difficult numerically. When we start with an initial mesh of $M_0 = 10$ subintervals in experiment #2, both MirkDC and PMirkDC choose the mesh selection sequence shown in Figure 4.46 and convergence is achieved on a final mesh of 2970 subintervals. Figure 4.44 shows the overall speedup and execution time of MirkDC and PMirkDC in this experiment, and Figure 4.45 shows the speedup and execution time of each of the five parallelized computational tasks. The experiment is run on the Challenge L. Parallelism begins to pay off with 2 processors, as is reflected in both the overall execution time and individual task execution times.

In experiment #4, we solve the same problem on the Origin 2000, starting with the same initial mesh and using the same defect tolerance. The results are shown in Figures 4.47 and 4.48. Execution times are nearly 5 times as fast. Comparing Figures 4.47 and 4.44, we see less optimal speedup on the Origin 2000, and parallelism does not begin to pay off until 3 processors. The reason for the apparent degradation in speedup is that the problem is too easy, and is solved too quickly. As a result, non-parallelized overhead tasks become a noticeable contributor to overall execution time. Similar parallel performance degradation was observed in the experiments in §4.3.

In experiments #1 and #3, we have substantially increased the execution time required to solve Problem A by selecting an initial mesh of $M_0 = 7000$ subintervals. The results are shown in Figures D.19, D.20, D.21 and D.22 of Appendix D.2. In these experiments, we see much

Figure 4.45: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #2 of Table 4.10.

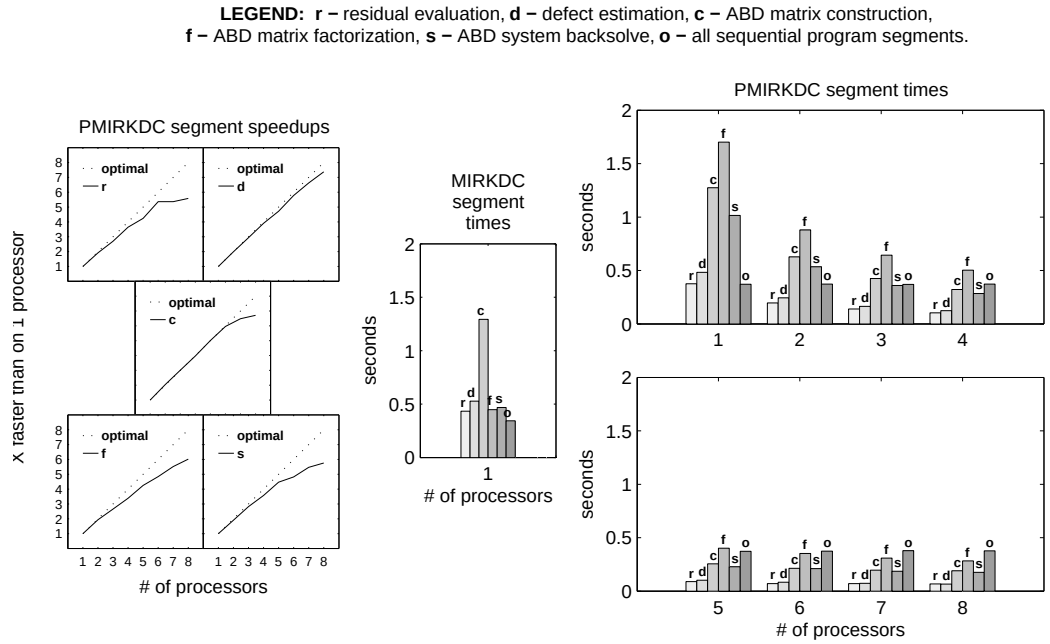


Figure 4.46: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #2 of Table 4.10.

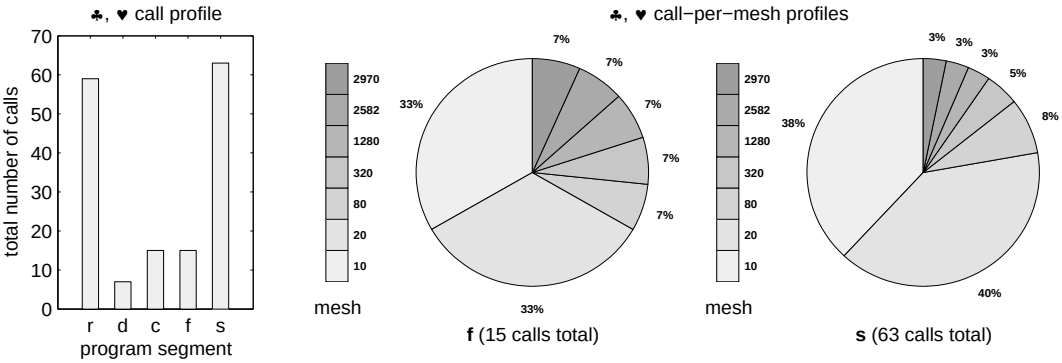


Figure 4.47: Overall speedup and execution time of MirkDC and PMirkDC in experiment #4 of Table 4.10. The same problem is solved as in experiment #2, using the same solution strategy, but this time on the Origin 2000 instead of the Challenge L. Execution times are nearly 5 times faster (compare to Figure 4.44). Parallelism begins to pay-off at 3 processors.

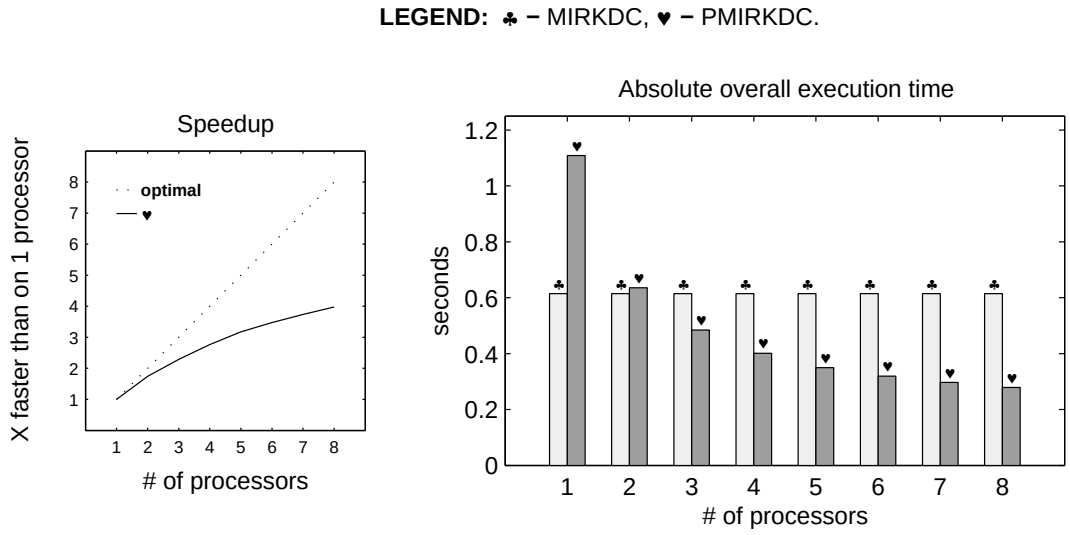


Figure 4.48: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #4 of Table 4.10. Compare to Challenge L results shown in Figure 4.45.

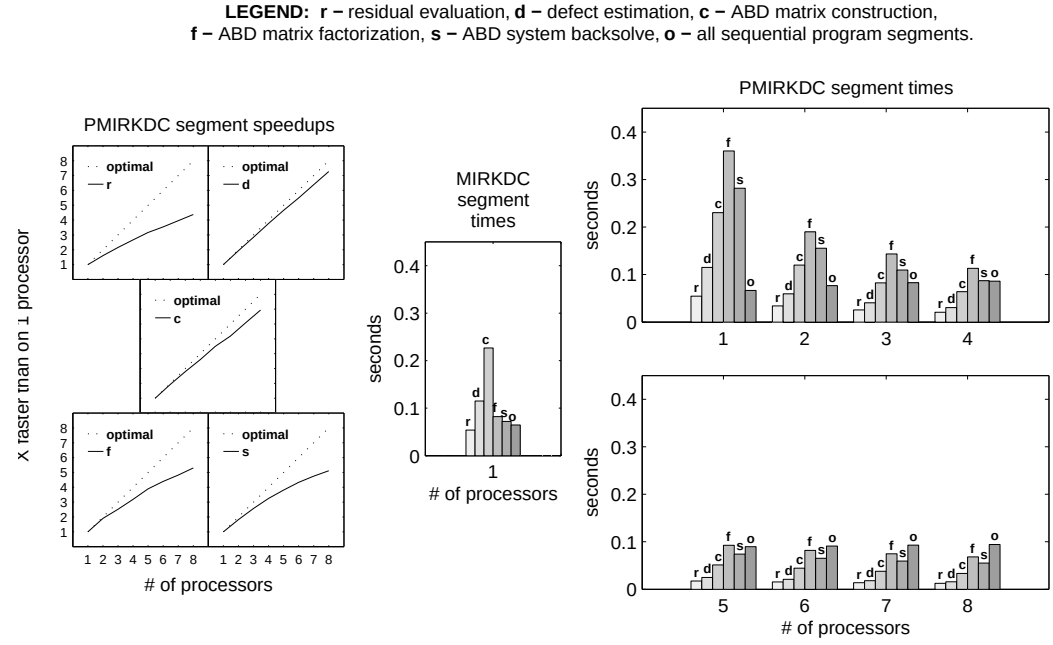
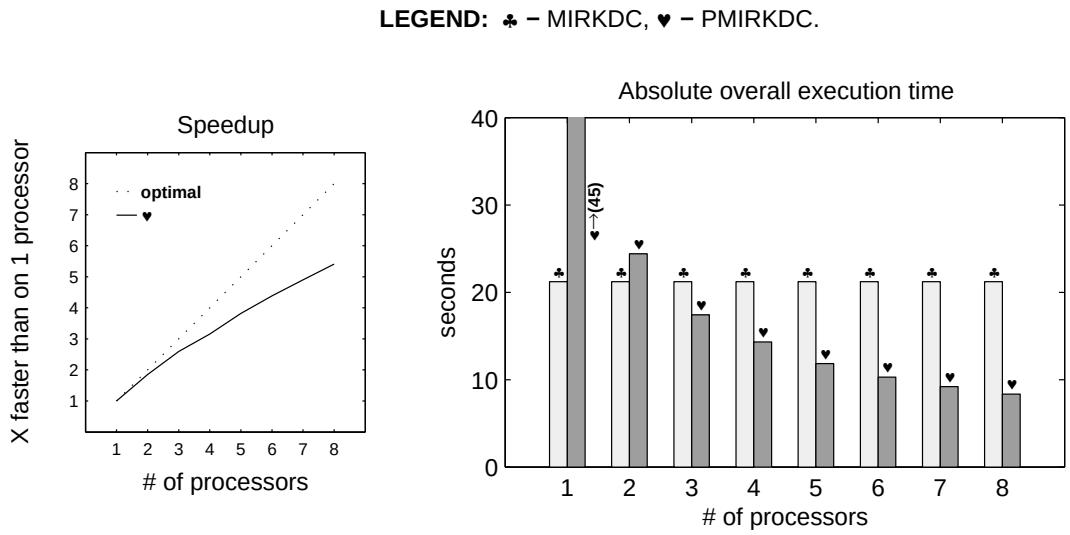


Figure 4.49: Overall speedup and execution time of MirkDC and PMirkDC in experiment #8 of Table 4.10. Results are shown for the final continuation step only. Parallelism begins to pay-off at 3 processors.



better speedups. In fact, Figure D.20 shows nearly perfect linear speedup in each task execution time when we run on the Challenge L, a phenomenon rarely seen in our experiments. These experiments, however, are not realistic. $M_0 = 7000$ is much too fine of an initial mesh for this problem. MirkDC immediately redistributes the mesh and computes an acceptable solution on a mesh of only 3744 subintervals.

We can create more realistic computationally intensive experiments by solving more difficult SWF-III problems, such as problems B, C, D and E of Table 4.9. We solve these in experiments #5, #6, #7 and #8, respectively. In each experiment, a parameter continuation strategy is adopted as discussed in §4.4.1. The results of experiments #5, #6 and #7 are shown in Figures D.23- D.31 of Appendix D.2. The results of experiment #8, the most computationally intensive of the four, are shown here in Figures 4.49-4.52.

Figure 4.52 shows the subroutine call and call-per-mesh profiles of MirkDC and PMirkDC accumulated over all five continuation iterations in experiment #8. Figure 4.51 shows the profiles for the final, most expensive, continuation step only. We see many more meshes, finer meshes, and many more task calls than in the experiments with Problem A. Figures 4.49 and 4.50 show overall and individual task execution times significantly greater—and speedups are closer to optimal—than in previous experiments run on the Origin 2000 (experiments #3 and #4).

Figure 4.50: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #8 of Table 4.10. Results are shown for the final continuation step only.

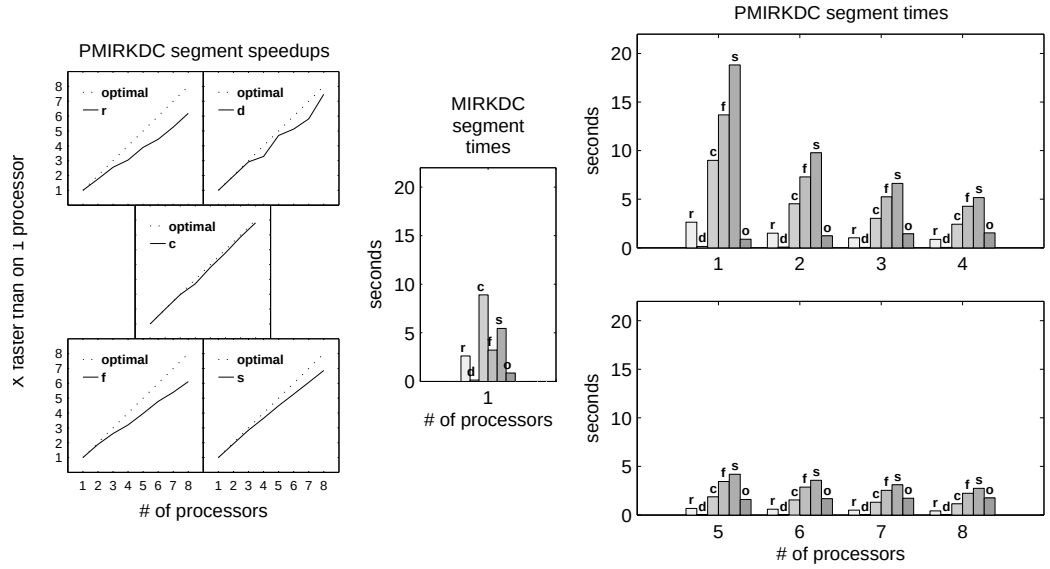


Figure 4.51: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #8 of Table 4.10. Results are shown for the final continuation step only, which is the most computationally intensive.

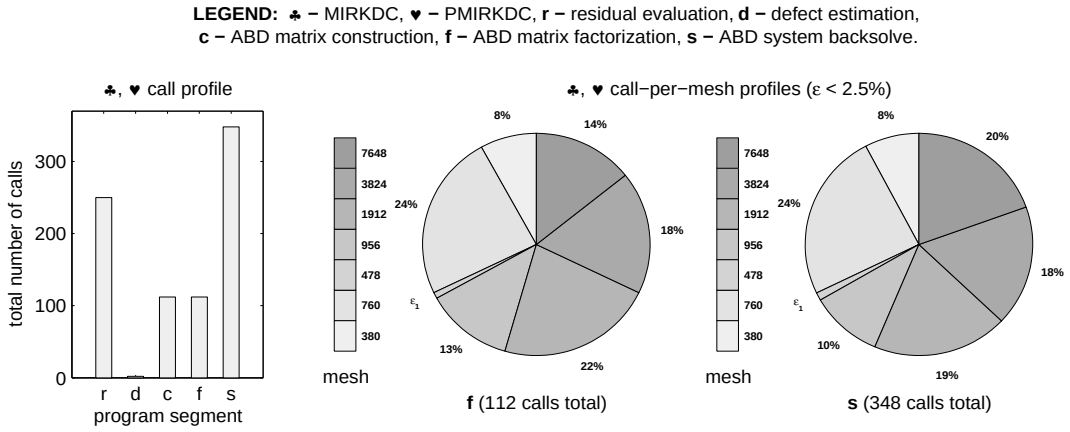
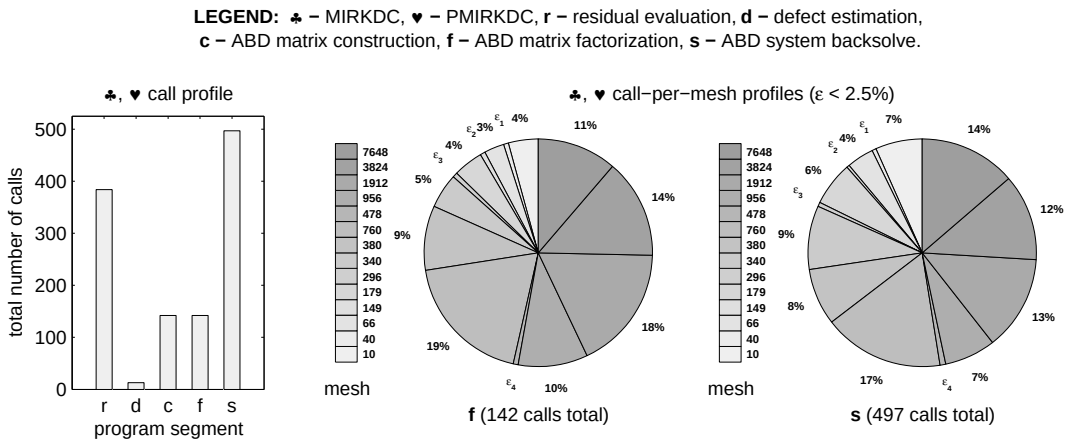


Figure 4.52: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #8 of Table 4.10. Results are accumulated over all five continuation steps.



When a difficult SWF-III problem is solved without using parameter continuation, MirkDC convergence may be slower, or it may not occur at all. We have also found that for some difficult problems, the MirkDC convergence pattern is affected by the choice ABD system solver. For the purposes of the experiments in this section, the convergence patterns of MirkDC and PMirkDC must be identical on a given problem in order to obtain a fair and accurate measure of the gains of parallelism. In particular, a longer, more expensive convergence pattern in PMirkDC could easily overshadow any improvement in execution time gained through parallelism.

What we have found, however, is the opposite. We have identified difficult problems for which the convergence pattern of PMirkDC is *shorter* than that of MirkDC when parameter continuation is not used. For example, in experiments #9 and #10 of Table 4.10 we attempt to solve Problems D and E directly for the given value of ϵ, without using parameter continuation. These experiments are run on the Sun Ultra 2, with PMirkDC in sequential, single-partition mode. The resulting convergence patterns of MirkDC/PMirkDC are shown in Figures D.34/D.35 and D.36/D.37 of Appendix D.2, respectively. In both experiments, PMirkDC converges on fewer, coarser meshes, with fewer task calls than MirkDC, which explains the improvement in performance reflected in both the overall execution time and individual task execution times shown in Figures D.32 and D.33. At the time of writing, the reason for the shorter convergence patterns is unknown. It is also unknown if problems exist where the reverse occurs; i.e., where the convergence pattern of PMirkDC is longer than that of MirkDC. Preliminary analysis, though, indicates that RSCALE may be acting as a mild preconditioner

on some of the poorly-conditioned ABD systems that typically arise when solving these difficult problems. More experiments comparing the behaviour of all four variants of MirkDC on difficult SWF-III problems (i.e., MirkDC/COLROW, MirkDC/RSCALE, MirkDC/*SLF*-QR and MirkDC/*SLF*-LU) are discussed in §4.4.4.

4.4.3 Problems Where MirkDC/*SLF*-LU Fails

The analysis in §3.2 suggests that there may be a higher probability of *SLF*-LU failure when solving an ABD system arising from a discretization over a mesh with several wide subintervals. Such a system can arise in the discretization of SWF-III when the coaxial disks are spread far apart. In the resulting physical system, most of the fluid motion occurs close to either disk and there is little if any motion throughout much of the interior. In this case, an adaptive mesh selection strategy—such as the one used in MirkDC–often leads to a discretization over a mesh with many wide subintervals covering the interior of the interval of integration.

To this end, we have designed the eight MirkDC/SWF-III experiments shown in Table 4.11. Each experiment uses a wide interval of integration and large viscosity parameter ϵ. (ϵ cannot be too small in these problems if the solution is to be obtained in a reasonable amount of time.) Many of the ABD systems generated by MirkDC in these experiments have the characteristic mentioned above, and as a result could cause problems for *SLF*-LU. Note that some of the experiments differ only in the number of partitions used—this turns out to be a key factor in determining *SLF*-LU stability.

Figure 4.53 shows the absolute overall execution time of the four variants of MirkDC in each experiment listed in Table 4.11. We see immediately that in all but experiment #6, MirkDC/*SLF*-LU uses significantly more execution time than the other variants of MirkDC. More importantly, MirkDC/*SLF*-LU does not converge in experiments #2, #4, #5 and #8.

In order to determine the cause of performance degradation in MirkDC/*SLF*-LU, we now look closely at experiments #1 and #2. (These experiments differ only in the number of partitions used.) Figure 4.54 shows the task execution times in experiment #1. The degradation in MirkDC/*SLF*-LU overall execution time is reflected in each of the five task execution times. The cause of the degradation is made clear by examining the convergence pattern in experiment #1. More precisely, we must examine *two* convergence patterns: one for the stable variants of MirkDC—MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/RSCALE—and one for the unstable variant—MirkDC/*SLF*-LU. These convergence patterns are shown in Figures 4.55 and 4.56, respectively.

Table 4.11: Eight MirkDC/SWF-III experiments to show how *SLF*-LU instability can affect MirkDC performance. A SWF-III problem is defined by ϵ and the interval of integration $[t_a, t_b]$. A MirkDC solution strategy is specified by a MIRK scheme, defect tolerance τ_{defect}, number of initial mesh subintervals M_0, and number of partitions.

exp. #	SWF-III parameters		MirkDC solution strategy			
	ϵ	$[t_a, t_b]$	MIRK scheme	τ_{defect}	M_0	# part.
1	1	$[-100, 100]$	Lobatto, 4^{th} order	10^{-5}	10	1
2	1	$[-100, 100]$	Lobatto, 4^{th} order	10^{-5}	10	2
3	0.6	$[-100, 100]$	Lobatto, 4^{th} order	10^{-5}	10	1
4	0.6	$[-100, 100]$	Lobatto, 4^{th} order	10^{-5}	10	2
5	0.6	$[-100, 100]$	Lobatto, 4^{th} order	10^{-5}	10	3
6	0.4	$[-90, 90]$	6^{th} order	10^{-7}	10	1
7	0.4	$[-90, 90]$	6^{th} order	10^{-7}	10	2
8	0.4	$[-90, 90]$	6^{th} order	10^{-7}	10	3

Figure 4.53: Overall execution time of the four variants of MirkDC when run on the experiments specified in Table 4.11. In all but experiment #6, MirkDC/*SLF*-LU performance is degraded by *SLF*-LU instability. Note that MirkDC/*SLF*-LU does not converge (i.e. τ_{defect} is not satisfied) in experiments #2, #4, #5, and #8.

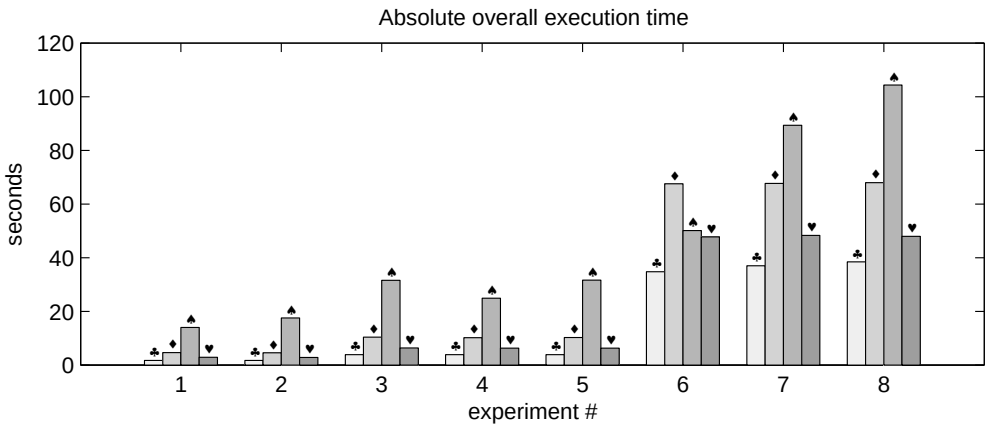


Figure 4.54: Overall and selected program segment execution times of the four variants of MirkDC in experiment #1 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures 4.55 and 4.56.

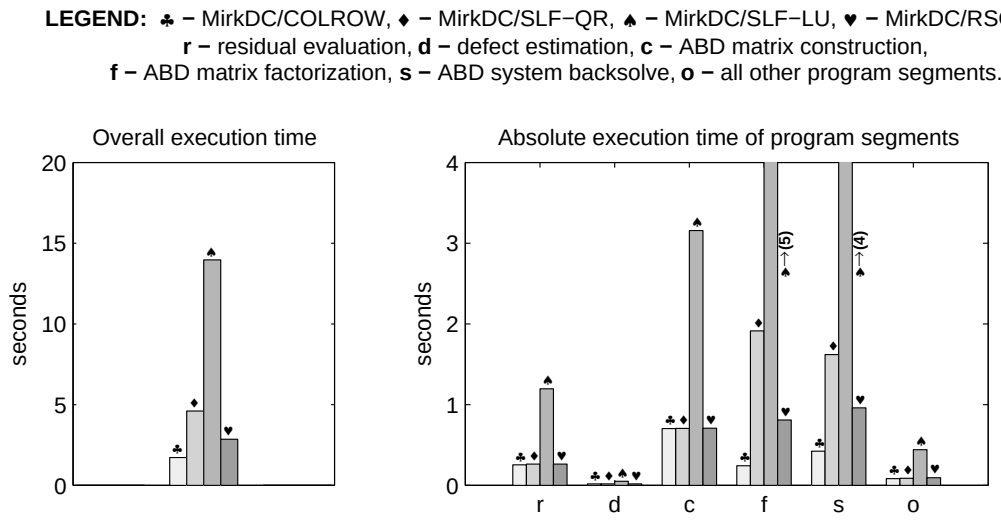


Figure 4.55: Profiles of the *stable* variants of MirkDC (MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/RSCALE) in experiments #1 and #2 of Table 4.11. Each of these variants converges in both experiments. Profiles are identical among variants and between experiments.

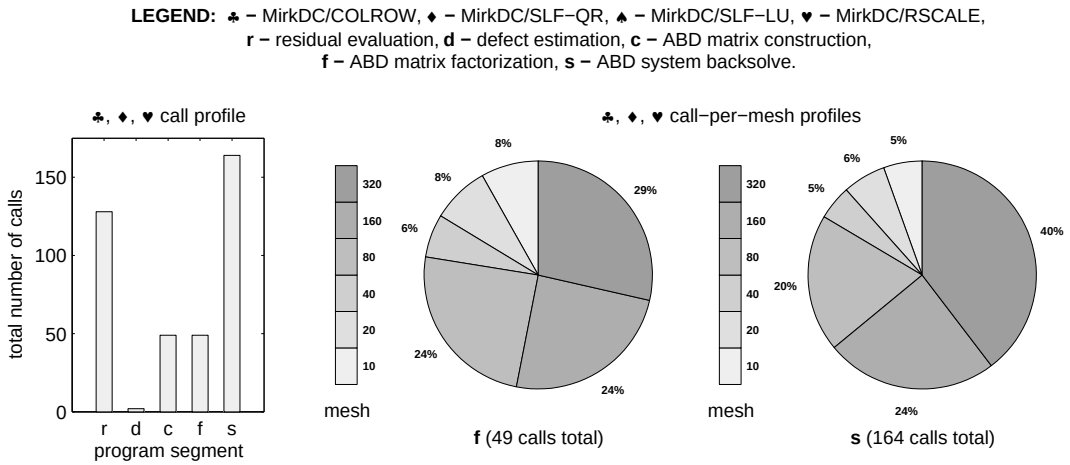


Figure 4.56: Profiles of MirkDC/*SLF*-LU in experiment #1 of Table 4.11. This variant of MirkDC converges in this experiment, albeit less efficiently than the other three variants (Figure 4.55). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 320 and 640. This instability is investigated further in Figure 4.57.

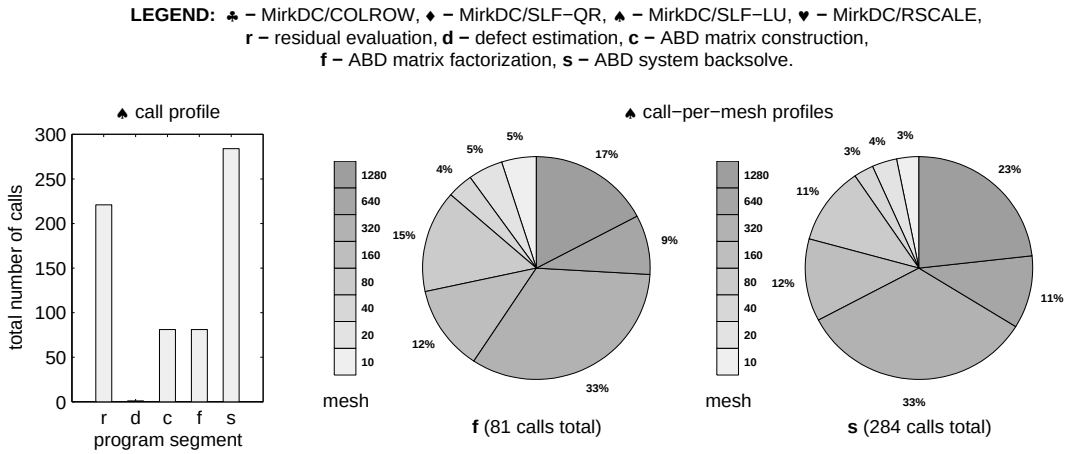
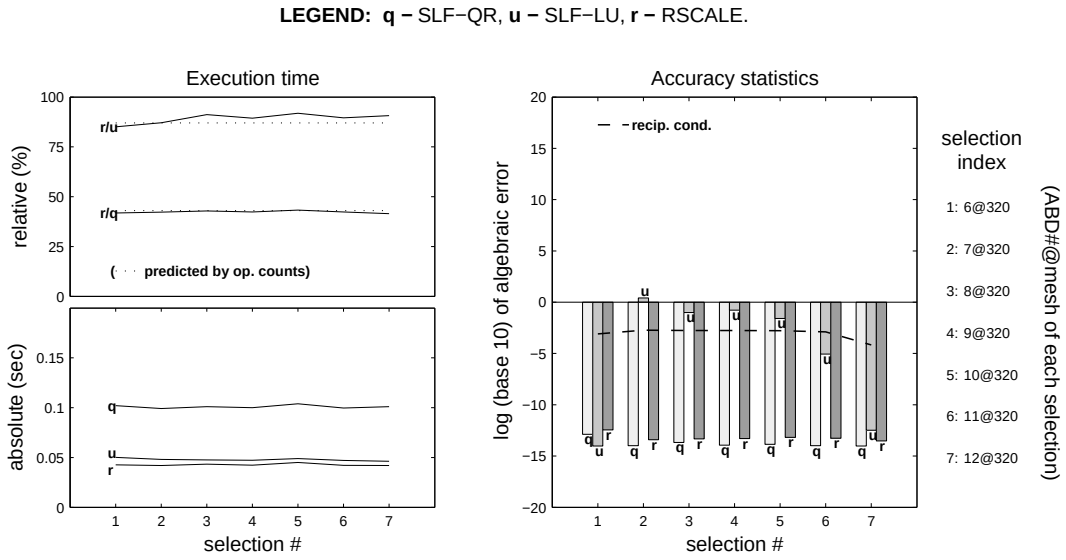


Figure 4.55 shows that MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/RSCALE exhibit identical convergence patterns in experiment #1, and that the convergence pattern does not change in experiment #2 when two partitions are used instead of one when solving the ABD systems. There are 49 factorizations in total, and the largest ABD system that arises consists of 321 block-rows. Figure 4.56 shows a notably different convergence pattern for MirkDC/*SLF*-LU. There are 81 factorizations in total, and the largest ABD system consists of 1281 block-rows. In fact, a significant number of such 1281 block-row systems arise—17% of 81 or 14 of them. It is clear why MirkDC/*SLF*-LU takes more time than the other variants of MirkDC in this experiment.

But why does MirkDC/*SLF*-LU exhibit a different convergence pattern? The reason lies in the stability of *SLF*-LU. To illustrate, we extract selected systems from the MirkDC/*SLF*-LU solution process in experiment #1, and solve these systems directly with each of the parallel ABD system solvers. The results are shown in Figure 4.57. We see that only *SLF*-LU exhibits instability when solving the 7^{th}, 8^{th}, 9^{th}, 10^{th} and 11^{th} ABD system built on a mesh of 320 subintervals. All systems are well-conditioned. (Note that the systems analyzed in Figure 4.57 are just an example. Several others arise during the solution process in experiment #1 that cause problems for *SLF*-LU.)

The accuracy of the ABD system solver inside of MirkDC has a direct effect on all other MirkDC tasks (defect estimation, mesh selection, etc.). Simply stated, the instability of *SLF*-

Figure 4.57: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #1 (Figure 4.56). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 7-11@mesh 320 (selection # 2-6).



LU in experiment #1 causes MirkDC to work harder in order to find an acceptable solution.

Figure 4.58 shows the task execution times in experiment #2. Again we see degradation in MirkDC/*SLF*-LU performance in each task, except for defect estimation (more on that below). Figure 4.59 shows the convergence pattern for MirkDC/*SLF*-LU in this experiment. (Recall that the convergence pattern for the other variants of MirkDC, shown in Figure 4.55, is unchanged from experiment #1.) MirkDC/*SLF*-LU computes 79 factorizations in total in experiment #2, and the largest ABD system consists of 10241 block-rows. ABD systems of 5121, 2561 and 1281 block-rows also arise, all much larger than in Figure 4.55. It is clear why MirkDC/*SLF*-LU takes more time than the other variants of MirkDC. We also note that MirkDC/*SLF*-LU does not converge. The code proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals (12000 in this implementation) is exceeded. The defect estimation stage is never reached.

The only difference between experiments #1 and #2 is the number of partitions used by the ABD system solvers. Comparing Figures 4.56 and 4.59, we see this had a notable effect on the convergence pattern of MirkDC/*SLF*-LU. As discussed in §4.4.2, one would hope that partitioning does not adversely affect convergence, because when it does any gain from parallelism could easily be negated.

Figure 4.58: Overall and selected program segment execution times of the four variants of MirkDC in experiment #2 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures 4.55 and 4.59.

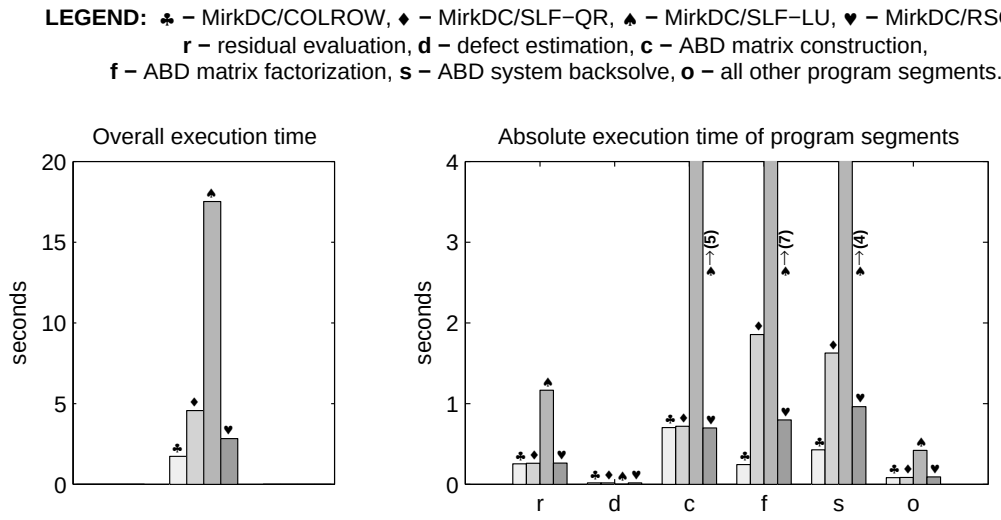
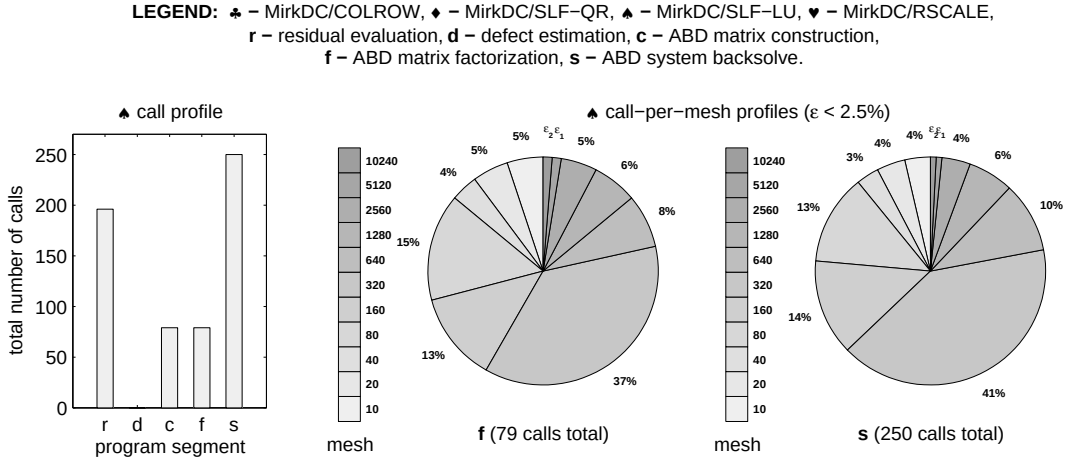


Figure 4.59: Profiles of MirkDC/*SLF*-LU in experiment #2 of Table 4.11. This variant of MirkDC does not converge in this experiment (i.e. τ_{defect} is not satisfied). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 320, 640, 1280, 2560, 5120 and 10240. This instability is investigated further in Figures 4.60, 4.61 and 4.62.



The failure of MirkDC/*SLF*-LU in experiment #2 again may be attributed to the instability of *SLF*-LU. In Figure 4.60, we extract selected systems from the MirkDC/*SLF*-LU solution process in experiment #2, and solve these systems directly with each of the parallel ABD system solvers. Only *SLF*-LU exhibits instability when solving ABD systems 6-9@mesh 320, 6@mesh 640, 5@mesh 1280, 3-4@mesh 2560 and 1@mesh 5120. All of these systems are well-conditioned, and there are several others that arise during the solution process in experiment #2 that cause problems for *SLF*-LU.

The effect of partitioning on the accuracy of *SLF*-LU is investigated further in Figures 4.61 and 4.62. In Figure 4.61, we extract ABD system 5@mesh 1280 and solve it directly, using 1-8 partitions, with each of the parallel ABD system solvers. Only *SLF*-LU exhibits instability when using 2-8 partitions. Note that non-partitioned *SLF*-LU is *stable* on this problem. This is somewhat surprising given the analysis in §3.2 which suggests that *SLF*-LU instability is likely to worsen as the size of the partition increases. In Figure 4.62, we extract ABD system 4@mesh 1280 and solve it directly using 1-8 partitions. Only *SLF*-LU exhibits mild instability when using 2, 4, 6 and 8 partitions, and yet is stable using 1, 3, 5 and 7 partitions. We note that an ABD matrix arising in the MirkDC solution process typically has variable block-rows (i.e., the blocks change from row to row). It seems that for these types of systems, *SLF*-LU stability may depend not only on partition size, but also on breakpoint positioning.

The remaining experiments in Table 4.11 (#3-#8) are discussed in Appendix D.3.

4.4.4 Problems Where Sequential MirkDC/RSCALE Outperforms MirkDC/COLROW

When comparing the performance of parallel MirkDC/RSCALE (PMirkDC) and sequential MirkDC/COLROW (MirkDC) in §4.4.2 and [Muir 03], we found that for some difficult SWF-III problems the convergence patterns of the two codes differed. In particular, in experiments #9 and #10 in §4.4.2 the convergence pattern of PMirkDC was shorter than that of MirkDC, resulting in a substantial improvement in execution time even on a sequential machine. In this section we investigate other difficult SWF-III problems that cause differences in convergence patterns, this time comparing the performance of all four sequential variants of MirkDC on the problems—MirkDC/COLROW, MirkDC/RSCALE, MirkDC/*SLF*-QR and MirkDC/*SLF*-LU.

Table 4.12 lists eight experiments on difficult SWF-III problems. Each of these problems is difficult because of the magnitude of ϵ, not because of the length of the interval of integration. In each experiment, the convergence pattern of at least one variant of MirkDC differs from

Figure 4.60: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #2 (Figure 4.59). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 6-9@mesh 320, 6@mesh 640, 5@mesh 1280, 3-4@mesh 2560 and 1@mesh 5120.

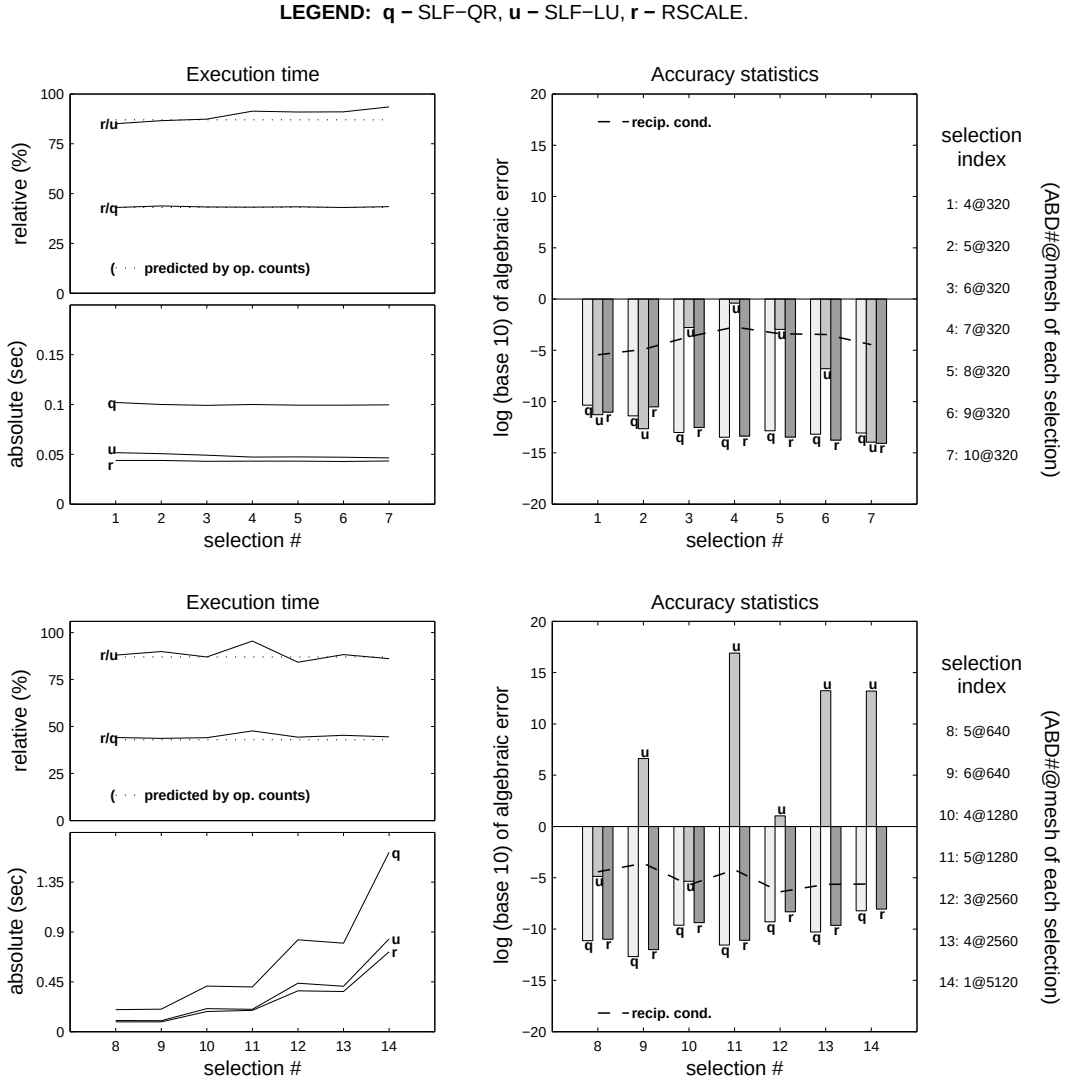


Figure 4.61: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 5@mesh 1280 extracted from the MirkDC/*SLF*-LU solution in experiment #2 (Figure 4.59). Of the three ABD solvers, only *SLF*-LU exhibits instability when using 2-8 partitions. Surprisingly, non-partitioned *SLF*-LU is *stable* on this problem.

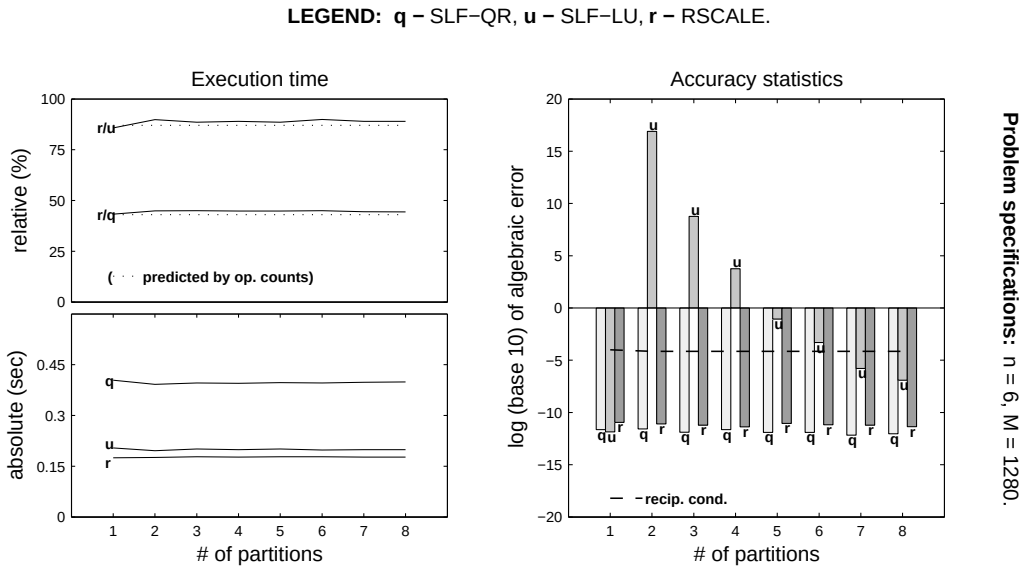


Figure 4.62: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 4@mesh 1280 extracted from the MirkDC/*SLF*-LU solution in experiment #2 (Figure 4.59). Of the three ABD solvers, only *SLF*-LU exhibits mild instability when using 2,4,6 and 8 partitions. Surprisingly, *SLF*-LU is *stable*—and also the most accurate of the three solvers—when using 1, 3, 5 and 7 paritions.

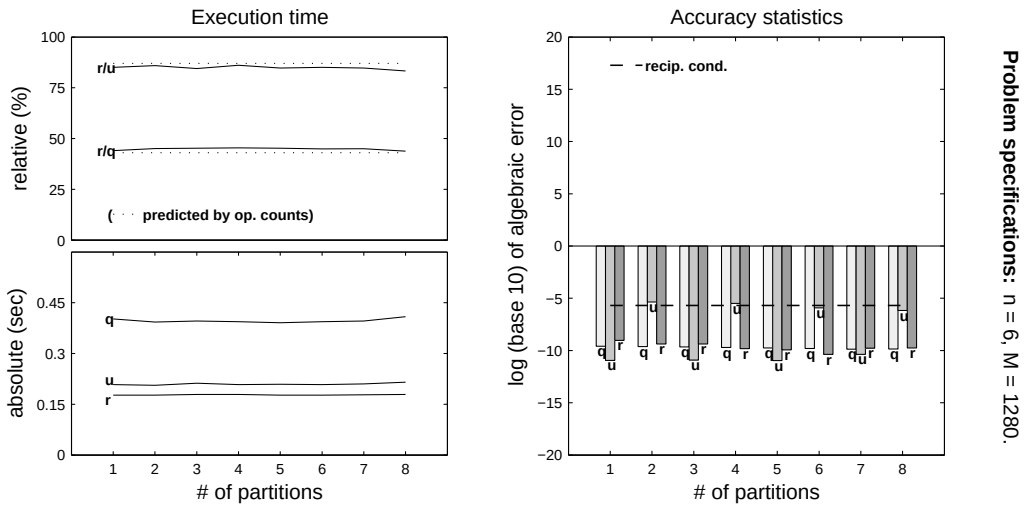
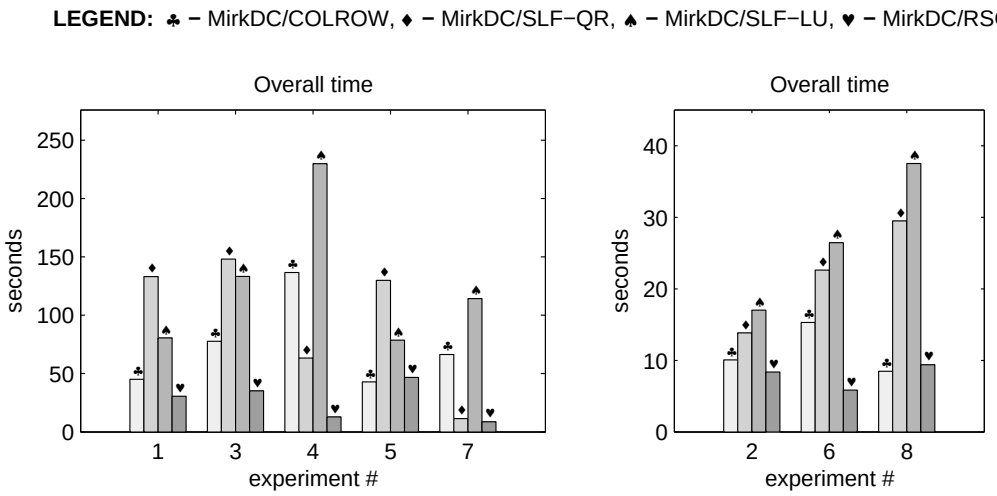


Table 4.12: Eight MirkDC/SWF-III experiments in which sequential MirkDC/RSCALE outperforms MirkDC/COLROW in terms of overall execution time and/or storage requirements. A SWF-III problem is defined by ϵ and the interval of integration $[t_a, t_b]$. A MirkDC solution strategy is specified by a MIRK scheme, defect tolerance τ_{defect}, number of initial mesh subintervals M_0, and number of partitions.

exp. #	SWF-III parameters		MirkDC solution strategy			
	ϵ	$[t_a, t_b]$	MIRK scheme	τ_{defect}	M_0	# part.
1	0.0012	$[-2, 2]$	trapezoidal, 2^{nd} order	10^{-4}	10	1
2	0.0007	$[-2, 2]$	Lobatto, 4^{th} order	10^{-6}	10	1
3	0.00045	$[-2, 2]$	Lobatto, 4^{th} order	10^{-7}	10	1
4	0.0003	$[-2, 2]$	6^{th} order	10^{-9}	10	1
5	0.00025	$[-1, 1]$	trapezoidal, 2^{nd} order	10^{-4}	10	1
6	0.000125	$[-1, 1]$	Lobatto, 4^{th} order	10^{-6}	10	1
7	0.0001	$[-1, 1]$	Lobatto, 4^{th} order	10^{-7}	10	1
8	0.000060	$[-1, 1]$	6^{th} order	10^{-9}	10	1

Figure 4.63: Overall execution time of the four variants of MirkDC when run on the experiments specified in Table 4.12. Convergence is achieved by each variant in each experiment except for MirkDC/*SLF*-LU in experiment #4. In all but experiments #5 and #8, MirkDC/RSCALE converges faster than the other variants, including MirkDC/COLROW. In experiments #5 and #8, MirkDC/COLROW is marginally faster.



the others. There are also cases where all four patterns differ, or match pairwise. As in the experiments in §4.4.2, the differences in convergence patterns are substantial enough to result in significant differences in performance among the variants.

Figure 4.63 shows the overall execution time of each of the four variants of MirkDC when run on the experiments specified in Table 4.12. Convergence is achieved by each variant in each experiment except for MirkDC/*SLF*-LU in experiment #4. In all experiments, we see a significant difference in overall execution time among the variants. In most experiments, MirkDC/RSCALE converges faster than the other variants, including MirkDC/COLROW.

These performance differences are explained by examining the convergence patterns and individual task execution times in each experiment. Figure 4.64 shows the overall and selected program segment execution times of the four variants of MirkDC in experiment #1 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments except ABD matrix factorization. Comparing the subroutine call and call-per-mesh profiles in Figures 4.65 and 4.66, we see that MirkDC/RSCALE converges on fewer, coarser meshes, with fewer task calls, than any of the other variants.

Figure 4.67 shows the overall and selected program segment execution times of the four variants of MirkDC in experiment #2. MirkDC/RSCALE outperforms MirkDC/COLROW

Figure 4.64: Overall and selected program segment execution times of the four variants of MirkDC in experiment #1 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments except ABD matrix factorization. Compare the subroutine call and call-per-mesh profiles in Figures 4.65 and 4.66.

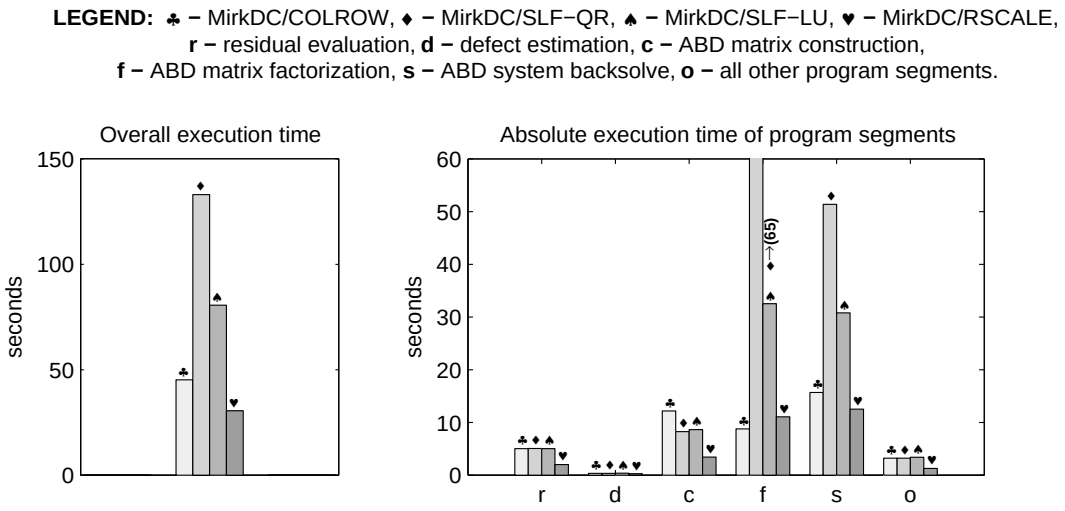


Figure 4.65: Profiles of MirkDC/COLROW and MirkDC/*SLF*-LU in experiment #1 of Table 4.12. Profiles of MirkDC/*SLF*-QR differ only slightly. These three variants of MirkDC converge in this experiment, but less efficiently than MirkDC/RSCALE (Figure 4.66).

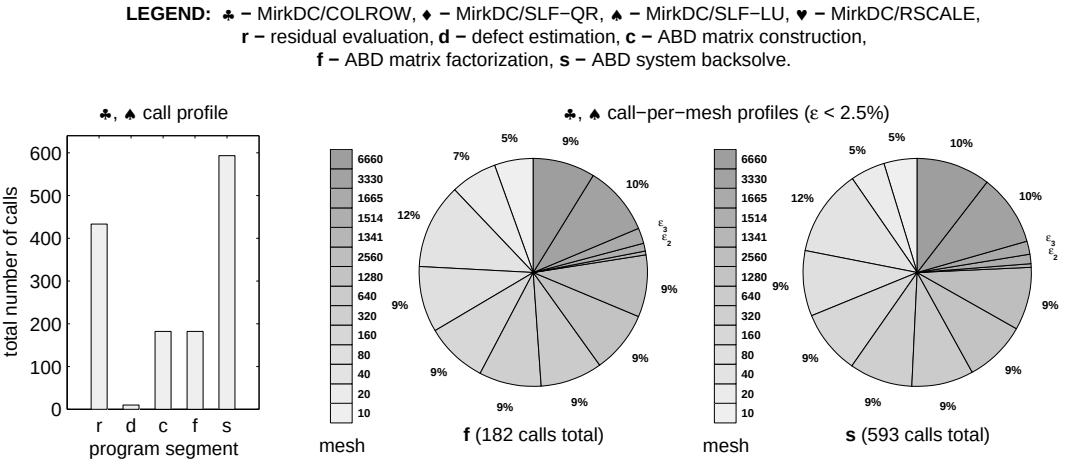
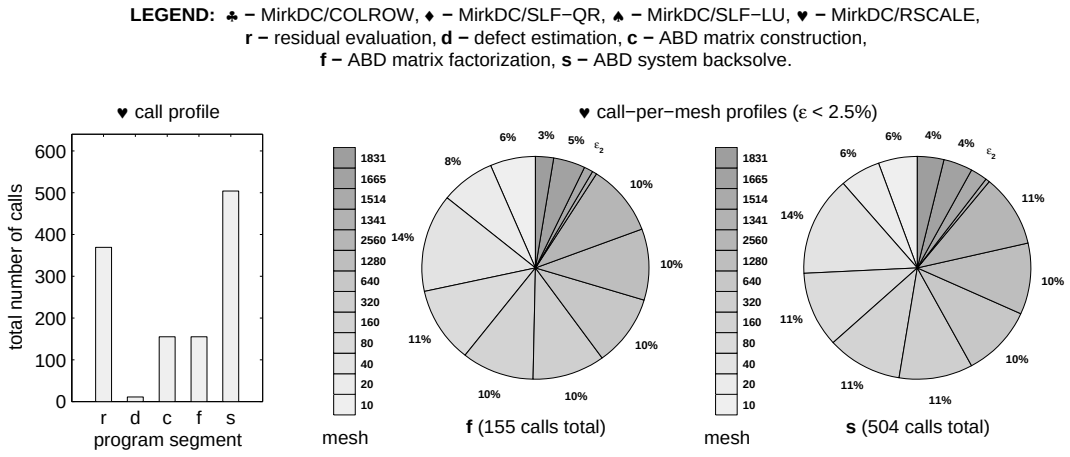


Figure 4.66: Profiles of MirkDC/RSCALE in experiment #1 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 1831. Compare these profiles to those of the other variants shown in Figure 4.65.



in all program segments except ABD matrix factorization and backsolve. In this experiment, MirkDC/RSCALE and MirkDC/*SLF*-QR have very similar convergence patterns, as do MirkDC/COLROW and MirkDC/*SLF*-LU. Comparing the subroutine call and call-per-mesh profiles in Figures 4.68 (MirkDC/COLROW, MirkDC/*SLF*-LU) and 4.69 (MirkDC/RSCALE, MirkDC/*SLF*-QR), we see that MirkDC/RSCALE and MirkDC/*SLF*-QR converge on fewer, coarser meshes, with fewer task calls, than both MirkDC/COLROW and MirkDC/*SLF*-LU. Since RSCALE is faster than *SLF*-QR, MirkDC/RSCALE outperforms MirkDC/*SLF*-QR in this experiment with respect to overall execution time.

The results for experiments #3-#8 of Table 4.12 are shown in Appendix D.4.

At the time of writing, the reason for the differences in convergence patterns among the variants of MirkDC is unknown. It is also unknown if problems exist where MirkDC/*SLF*-QR or MirkDC/*SLF*-LU outperform MirkDC/RSCALE. Preliminary analysis, though, indicates that RSCALE may be acting as a mild preconditioner on some of the poorly-conditioned ABD systems that typically arise when solving these difficult problems. We will pursue this possibility in future work.

Figure 4.67: Overall and selected program segment execution times of the four variants of MirkDC in experiment #2 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments except ABD matrix factorization and backsolve (backsolve times are nearly equal). Compare the subroutine call and call-per-mesh profiles in Figures 4.68 and 4.69.

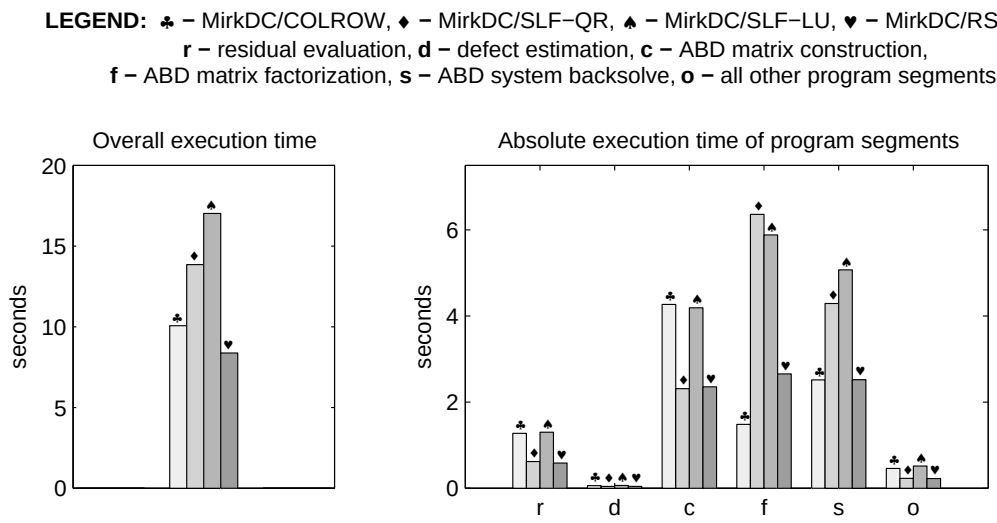


Figure 4.68: Profiles of MirkDC/COLROW and MirkDC/*SLF*-LU in experiment #2 of Table 4.12. These two variants of MirkDC converge in this experiment, but less efficiently than either MirkDC/RSCALE or MirkDC/*SLF*-QR (Figure 4.69).

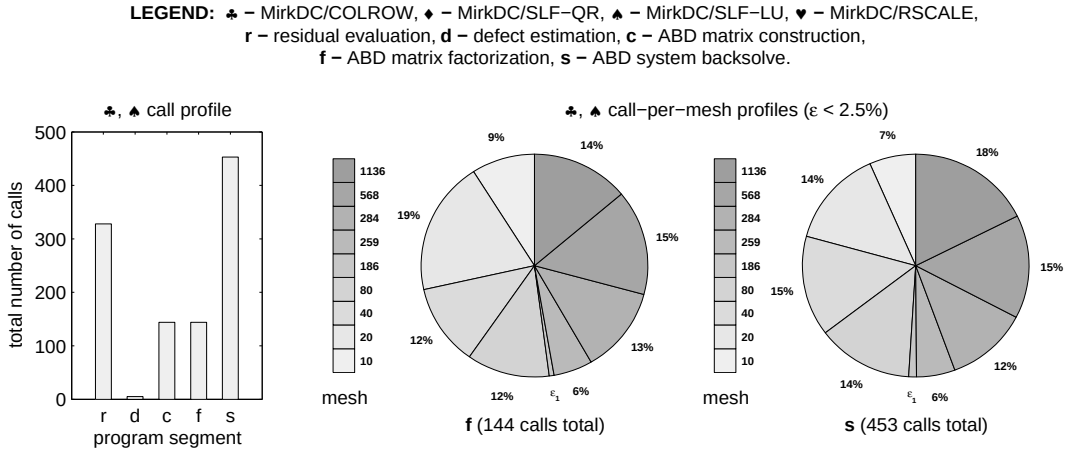
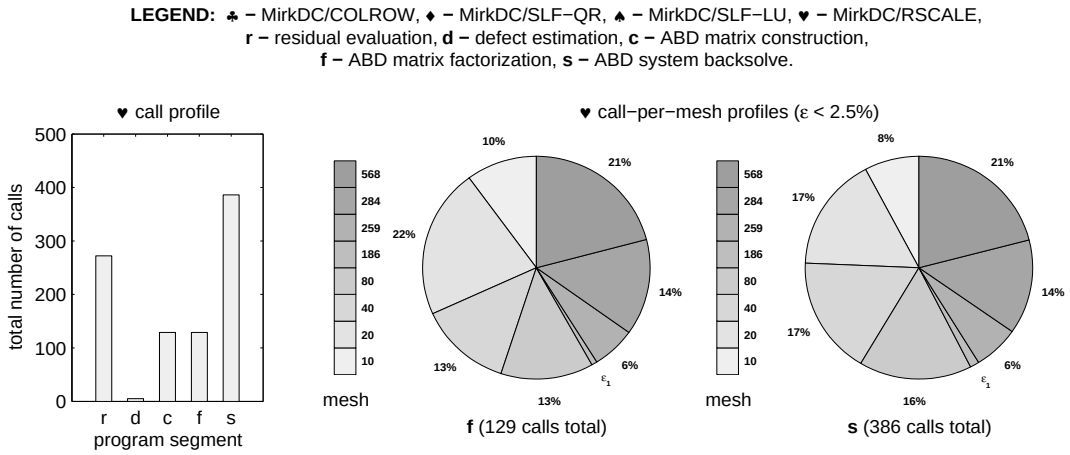


Figure 4.69: Profiles of MirkDC/RSCALE in experiment #2 of Table 4.12. Profiles of MirkDC/*SLF*-QR differ only slightly. These two variants of MirkDC converge in this experiment, with a final mesh of size 568. Compare these profiles to those of the other variants shown in Figure 4.68.



Chapter 5

Conclusions and Future Work

The factorization and solution of the ABD linear system constitute two of the most computationally intensive stages in a BVODE code. As other computationally intensive stages can be parallelized in a straightforward manner, there is a clear motivation for designing a parallel algorithm for treating the ABD linear system. Common partitioning or block-cyclic reduction approaches based on compactification obtain good speedup, but are potentially unstable on problems with rapidly increasing and/or decreasing fundamental solution modes. Since many well-posed BVODEs have this characteristic, these approaches are unsatisfactory. Our goal is to find a *stable*, parallel algorithm for solving the ABD linear system. We have proposed three such algorithms in this thesis—*SLF*-QR, *SLF*-LU and RSCALE.

Two of the algorithms—*SLF*-QR and *SLF*-LU– were discovered independently by us and by S.J. Wright in the 1990s. Wright presented these algorithms and analyzed their stability in [Wrig 92] and [Wrig 94]. We expand on the basic algorithms by proposing different variants that make better use of idle processors in order to more fully exploit parallelism. We do not attempt to expand on Wright's stability analysis in [Wrig 94], but we do identify a wider class of problems for which *SLF*-LU is potentially unstable. In [Wrig 92], Wright claims that *SLF*-QR is stable because it is equivalent to the QR-factorization of a row and column permuted version of the ABD matrix. We show the details of this equivalence for the single-partition variant of the algorithm. A similar argument may be used to show that *SLF*-LU is equivalent to a row-pivoted LU-factorization of a row and column permuted version of the ABD matrix. Although we do not include this in the thesis, if we apply this result to the additional problems we have identified for which *SLF*-LU is potentially unstable, we can identify an additional class of well-conditioned linear systems for which Gaussian elimination with row partial pivoting is unstable—systems different from those discussed in [Wrig 93].

As both *SLF*-QR and *SLF*-LU attain the theoretically optimal speedup for the problem if enough processors are available, not much can be done to improve on the global performance of these algorithms. There are other avenues for improvement, however. In particular, one could design an algorithm that uses fewer local operations per block-step, or has better stability properties than *SLF*-LU. To this end, we propose RSCALE—a third algorithm based on a notably different numerical technique. We show through extensive numerical testing and operation count analysis that RSCALE is approximately twice as fast as *SLF*-QR, is marginally faster than *SLF*-LU in most problems where *SLF*-LU pivots correctly to control stability, and is stable on problems where *SLF*-LU fails. We show that the complexity of the *SLF*-LU local factorization is dependent on its pivoting strategy, and our numerical tests suggest that in the majority of problems this complexity tends toward its upper bound. We give a preliminary analysis of the stability of RSCALE, point out some of its shortcomings, and address these shortcomings with a few simple modifications to the original algorithm.

We have carefully implemented each of *SLF*-QR, *SLF*-LU and RSCALE in FORTRAN, making extensive use of level-3 BLAS. We have tested the codes thoroughly on both a sequential and parallel machine, assessing the relative performance of the algorithms in terms of both accuracy and speed. In most cases, the relative execution time of the solvers agrees with the ratio predicted by the high-order coefficients of their respective operation counts. Tests on the parallel machine show that parallelism begins to pay off with just a few processors when we compare execution time to that of the best sequential solver for the problem. We also assess the performance of the solvers when they are incorporated in MirkDC [Enri 96], a software package for solving nonlinear BVODEs. The differences in speed and accuracy amongst the solvers is reflected in the overall performance of MirkDC, re-enforcing the importance of the role of the ABD system solver in BVODE software.

During the writing of this thesis, we contributed to the development of PMirkDC [Muir 03], a parallel implementation of MirkDC using RSCALE in place of COLROW as the ABD system solver. In this work, we targeted a tightly-coupled, shared memory architecture in our implementation. In future work, we hope to implement PMirkDC on a distributed-memory architecture, which has become more popular in recent years.

There are at least two possible contributions to sequential algorithms arising from the work in this thesis. First, during our numerical testing with MirkDC, we noticed that the choice of ABD system solver can have an effect on the convergence properties of the code. In particular, MirkDC/RSCALE converged faster than other variants of MirkDC (even MirkDC/COLROW) on some difficult problems. We believe RSCALE may act as a mild preconditioner on some

of the poorly-conditioned ABD systems that typically arise when solving these difficult problems. We are particularly interested in determining if the choice of relaxation parameter σ can improve the conditioning of the ABD matrix. (We know it can do so for the individual blocks; see §3.3.) We will pursue this possibility in future work. Second, as none of the parallel ABD system solvers require the boundary blocks of the linear system to be separable, problems with coupled boundary conditions could be handled directly in a BVODE code if we replace its sequential solver with one of our parallel solvers. This could result in substantial savings in a sequential implementation. The usual approach to handling such problems is to rewrite the BVODE with separated boundary conditions by doubling the number of differential equations, which in turn doubles the order of the ABD system and significantly increases the computational cost of most components in the code.

Another open question for future investigation follows from the observation that most of the examples of potential *SLF*-LU instability discussed in this thesis apply only to the single-partition variant of the algorithm, where the ABD matrix is processed in a sequential fashion from top to bottom. As shown in §3.2, the single-partition variant is easy to analyze, and with our analysis we gain some insight into where to look for instability. Other variants of *SLF*-LU may have different numerical properties. We give a few examples where the partitioned variant exhibits instability as the number of partitions changes (e.g., Figures D.57-D.60 in Appendix D.3), but at the time of writing we cannot comment further on these results. In addition, our preliminary testing shows that the cyclic-reduction variant of *SLF*-LU does not seem to exhibit the same instability on the class of problems we have identified to cause difficulty for the single-partition variant of the algorithm. This is not surprising, since with cyclic reduction the left and right blocks of the matrix change at each sweep of the reduction regardless of whether pivoting is used. (The change in block structure also makes the algorithm more difficult to analyze.) The stability of the cyclic-reduction variant of *SLF*-LU currently is in question, and we intend to pursue this in future work.

Appendix A

Additional σ/σ_k-RSCALE Experiments

This appendix contains additional numerical experiments run on Problems C-F of Table 3.1. As described in §3.3.2, each problem is constructed in such a way that static 1.0-RSCALE fails at the computation of one or more rescaled $\hat{V}_k$. See §3.3.2 for a complete description of how the problems are set up, and an explanation of how to interpret the error statistics and plots. The following table cross-references **Problem #** and **Algorithm** to give the figure and page # of each experiment in this appendix:

		Algorithm			
		1.0-RSCALE	0.98-RSCALE	1.02-RSCALE	σ_k-RSCALE
Problem #	C	A.1, p.167	A.2, p.168	A.3, p.169	A.4, p.170
	D	A.5, p.171	A.6, p.172	A.7, p.173	A.8, p.174
	E	A.9, p.175	A.10, p.176	A.11, p.177	A.12, p.178
	F	A.13, p.179	A.14, p.180	A.15, p.181	A.16, p.182

Of particular interest in these experiments are Problems E and F. In Problem E, selected eigenvalues ν_3 and ν_4 of V (i.e. r_{107} and $\bar{r}_{107}$) not only satisfy (3.82), but also (3.84) for $c = 4/5$, $3/5$, $2/5$ and $1/5$. (These particular 125-th roots of unity also happen to be 100-th, 75-th, 50-th and 25-th roots of unity.) This leads to additional singularities in the computation of $\hat{V}_{k^\dagger}$, $\tilde{V}_{k,k^\dagger-1}$ and $\tilde{W}_{k^\dagger,i}$ at $k^\dagger = 29$, 54, 79 and 104. Problem F demonstrates the ill-effects caused by a singularity in the computation of $\hat{V}_1$; namely a poorly-conditioned compacted matrix C and instability in the computation of $\tilde{y}_k$ in (3.16) (bottom left plot). Finally, note that in each of the σ_k-RSCALE solutions, only a few adjusted eigenvalue shifts are required to avoid the singularities arising in static 1.0-RSCALE, supporting our conjecture that ω_σ normally does not grow too large in this algorithm.

Figure A.1: The static 1.0-RSCALE solution to Table 3.1/C.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 1e-14	6.5e+15	6.5e+15	1.3e+30	0.00098	4.6e+12	7.5e+04	4.6e+12

The true value of $\mathcal{K}_2(C)$ is 94. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

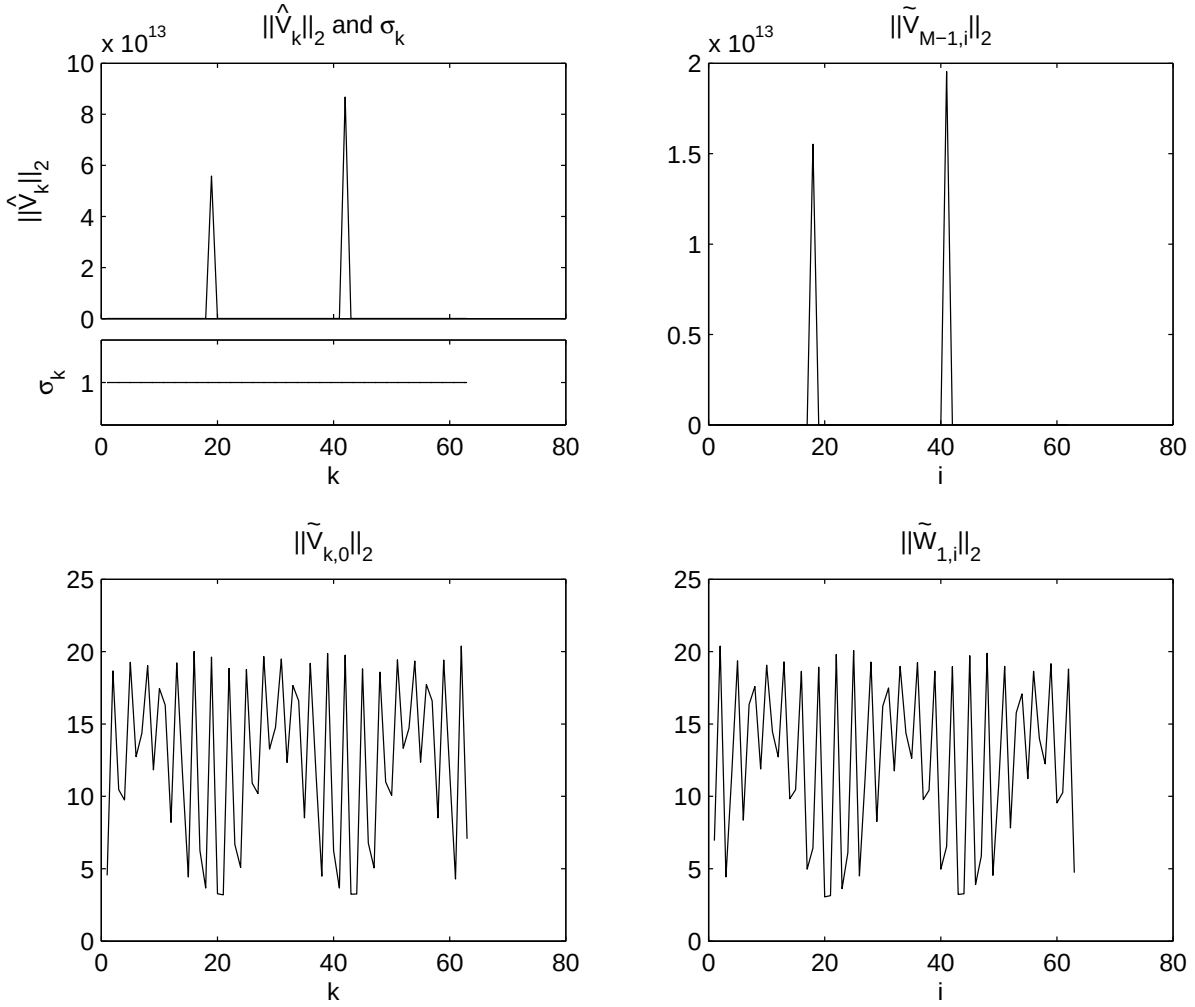


Figure A.2: The static 0.98-RSCALE solution to Table 3.1/C.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.6, 0.02	1.3e+04	3.4e+03	4.5e+06	0.00098	0.00053	7.5e+04	1.2e-11

The true value of $\mathcal{K}_2(C)$ is 1.5e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

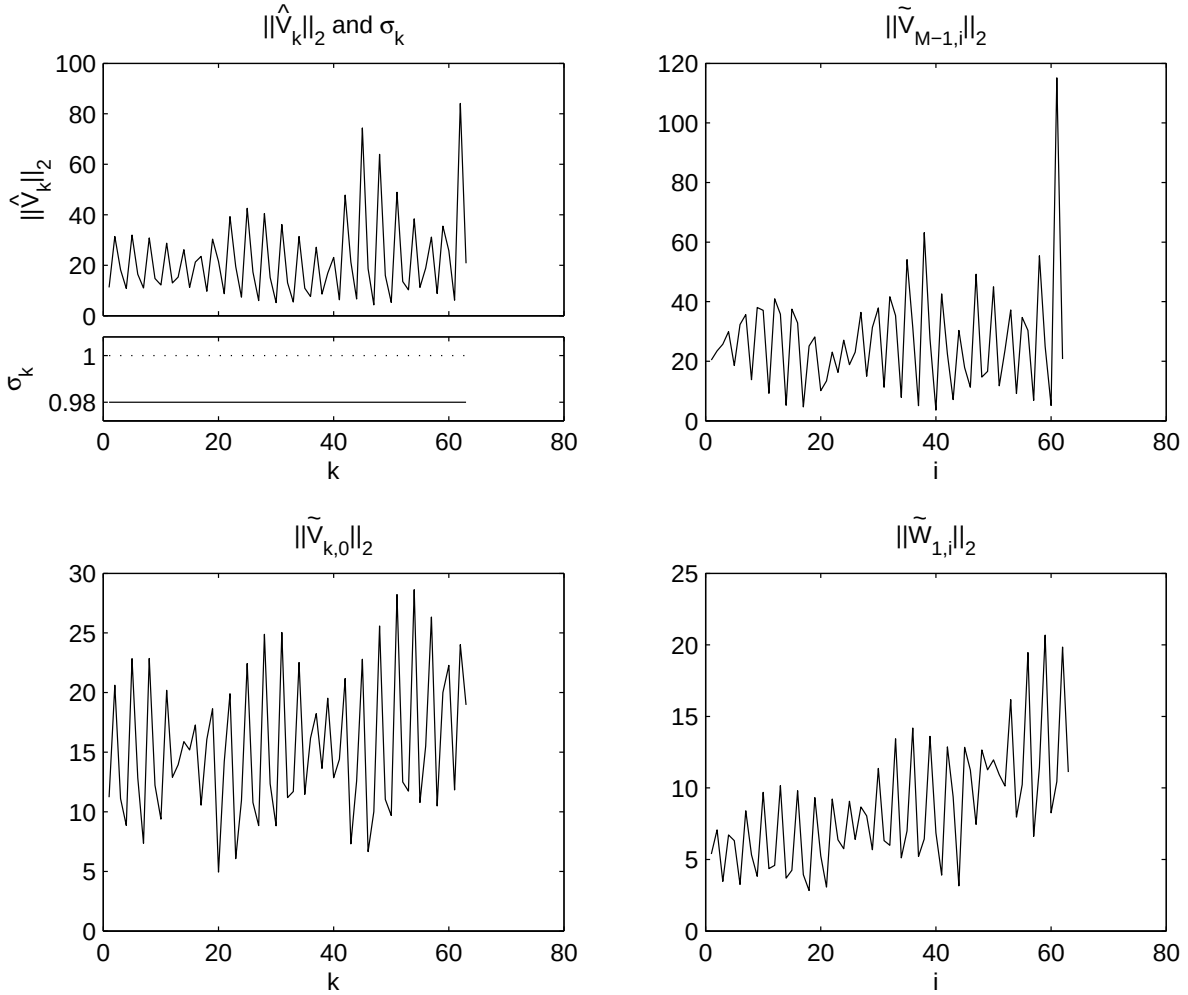


Figure A.3: The static 1.02-RSCALE solution to Table 3.1/C.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.6, 0.02	3.4e+03	3.4e+03	4.3e+06	0.00098	0.00053	7.5e+04	5.7e-11

The true value of $\mathcal{K}_2(C)$ is 1.2e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

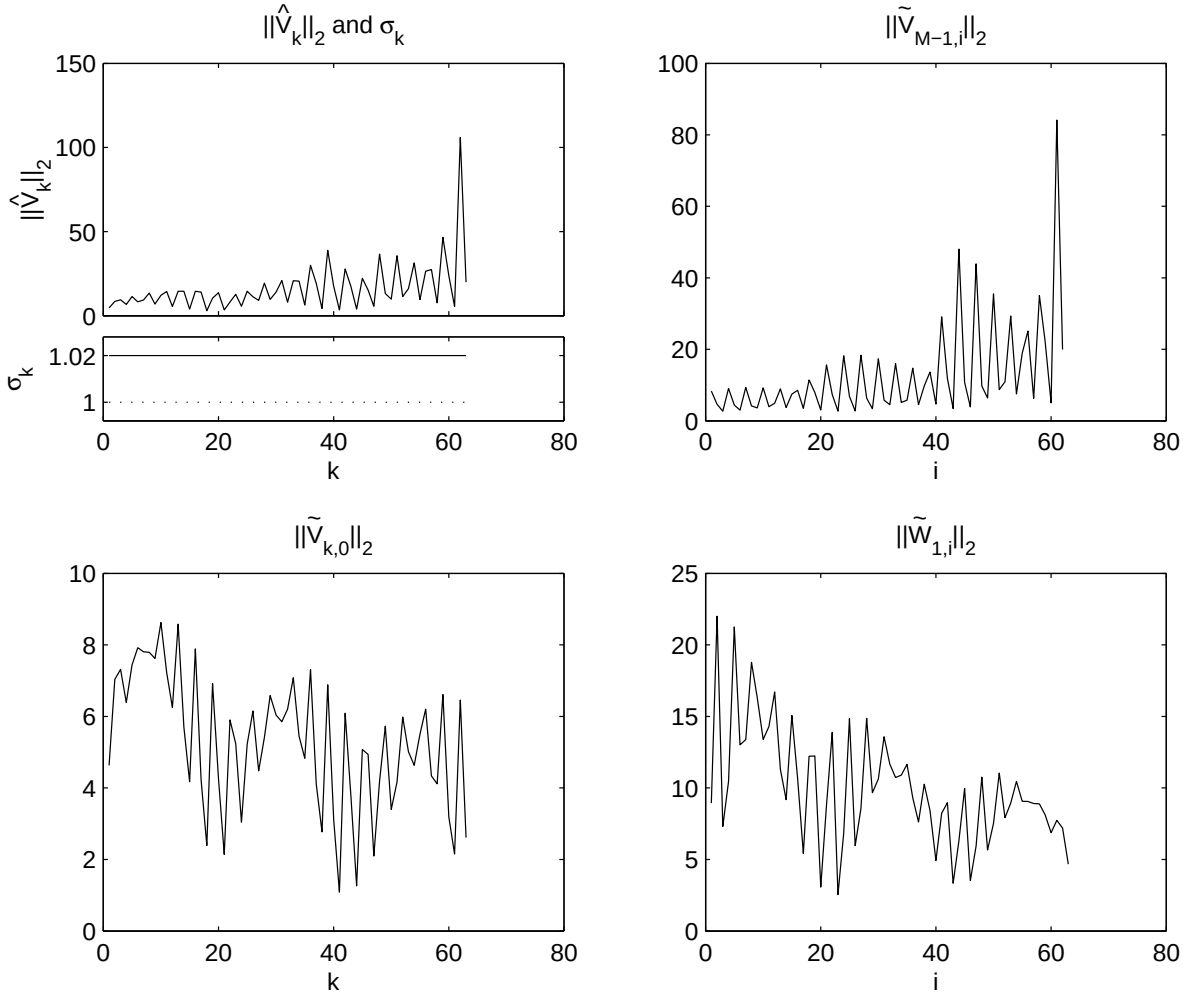


Figure A.4: The σ_k-RSCALE solution ($\tau_{\hat{V}} = 10.0,\ \epsilon_\sigma = 0.25$) to Table 3.1/C.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
9.9, ?	?	?	?	0.00098	0.00053	7.5e+04	5.1e-11

The true value of $\mathcal{K}_2(C)$ is 1.3e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.89) (bottom right):

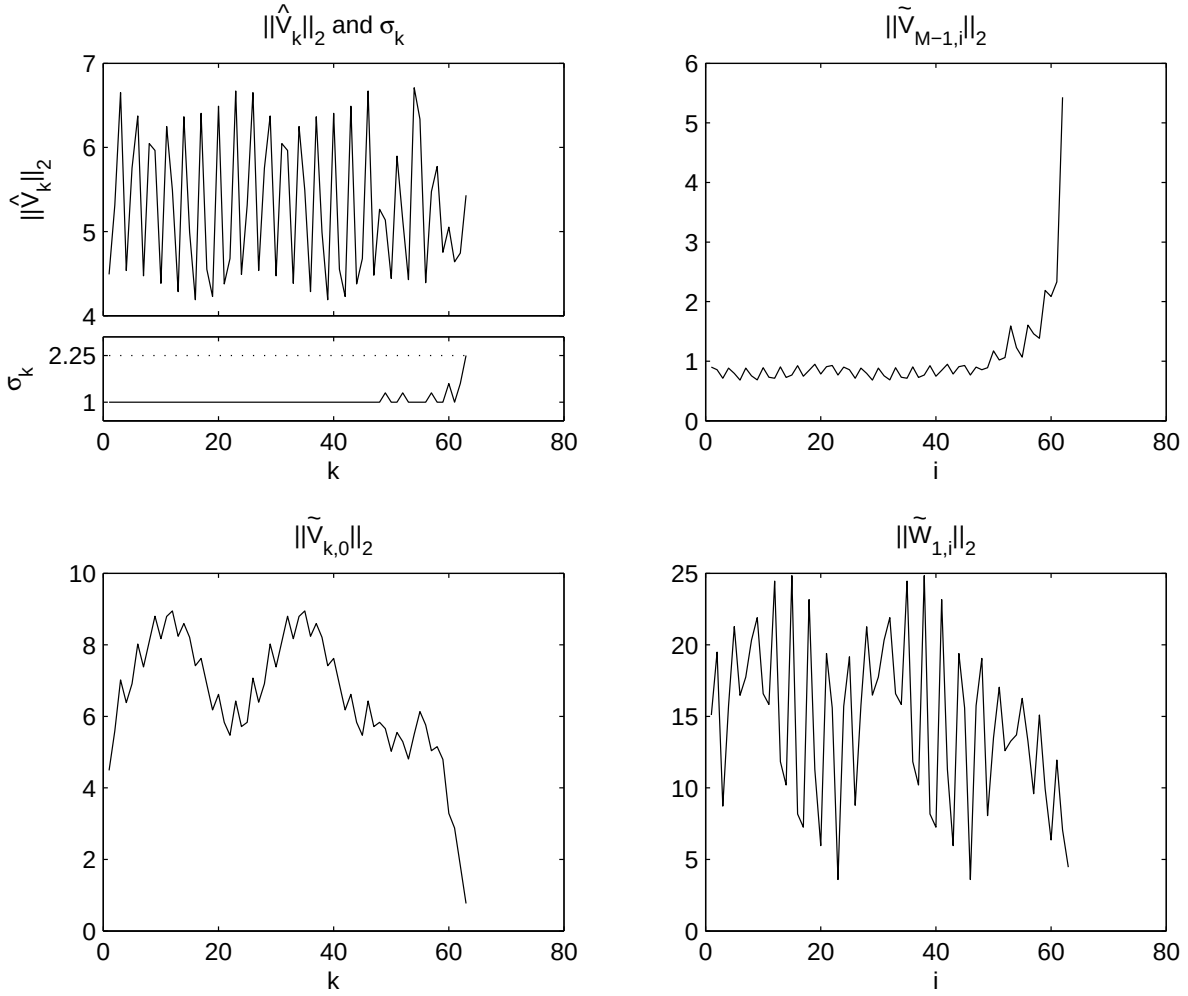


Figure A.5: The static 1.0-RSCALE solution to Table 3.1/D.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 1.1e-15	1.1e+17	1.1e+17	2e+32	0.0039	5.4e+12	3.9e+04	5.4e+12

The true value of $\mathcal{K}_2(C)$ is 57. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

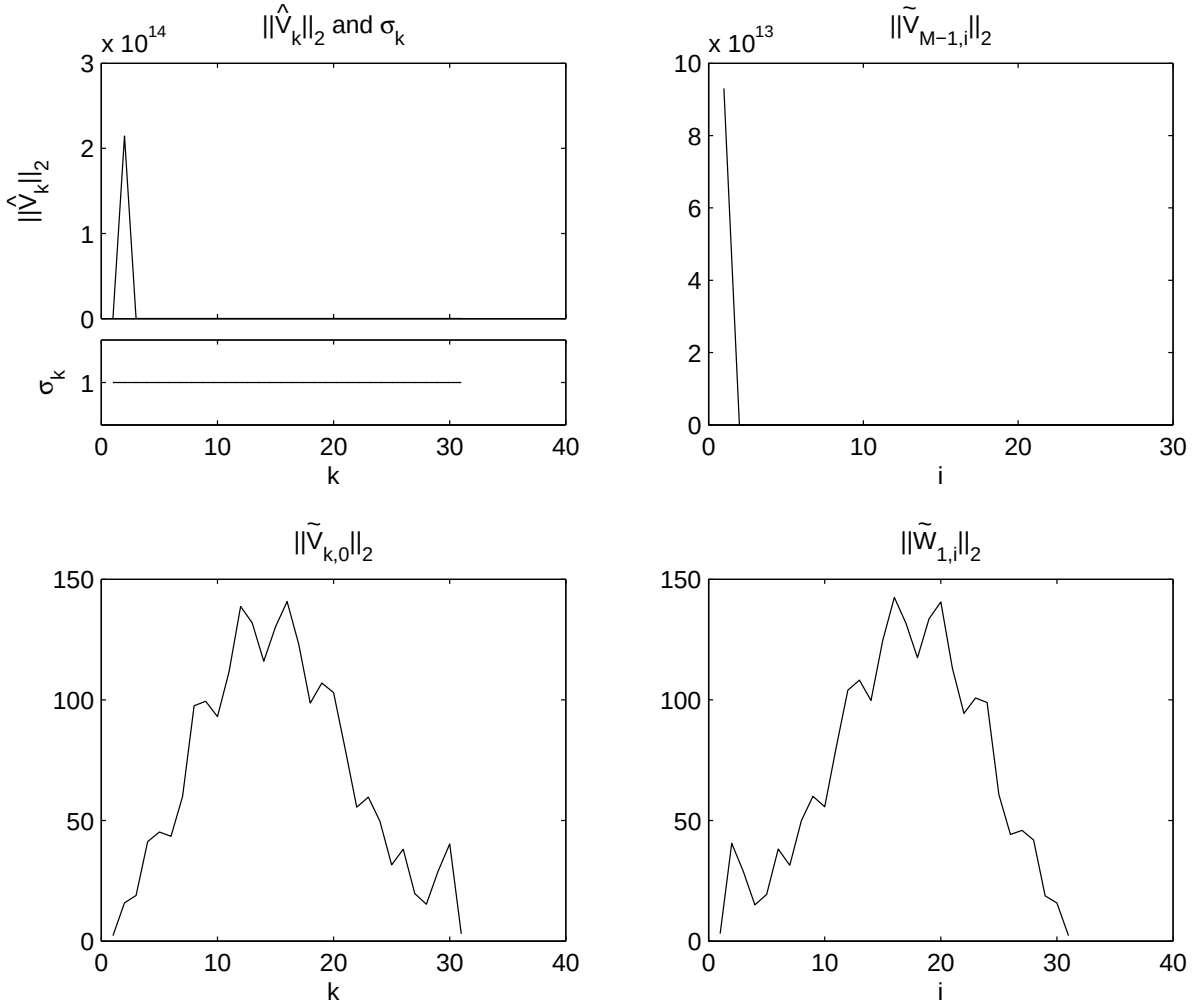


Figure A.6: The static 0.98-RSCALE solution to Table 3.1/D.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\|\tilde{V}_{k,i}\|\|_2$	$\|\|\tilde{W}_{k,i}\|\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1.9, 0.02	1.2e+04	6.3e+03	2.3e+06	0.0039	0.0023	3.9e+04	2.2e-11

The true value of $\mathcal{K}_2(C)$ is 1.1e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

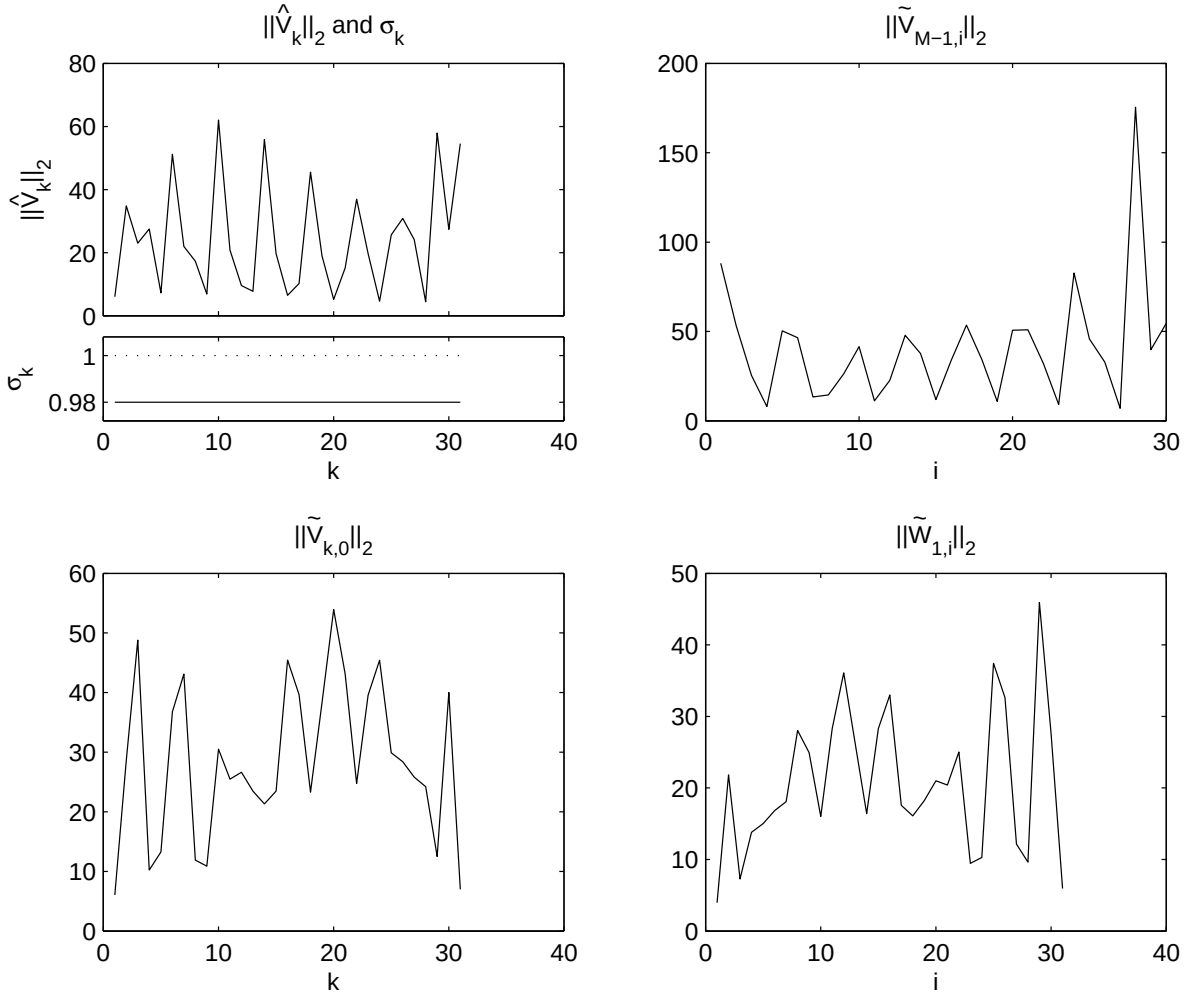


Figure A.7: The static 1.02-RSCALE solution to Table 3.1/D.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
ω_σ, ω_λ	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1.9, 0.02	6.3e+03	6.3e+03	2.2e+06	0.0039	0.0023	3.9e+04	2.4e-11

The true value of $\mathcal{K}_2(C)$ is 1e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

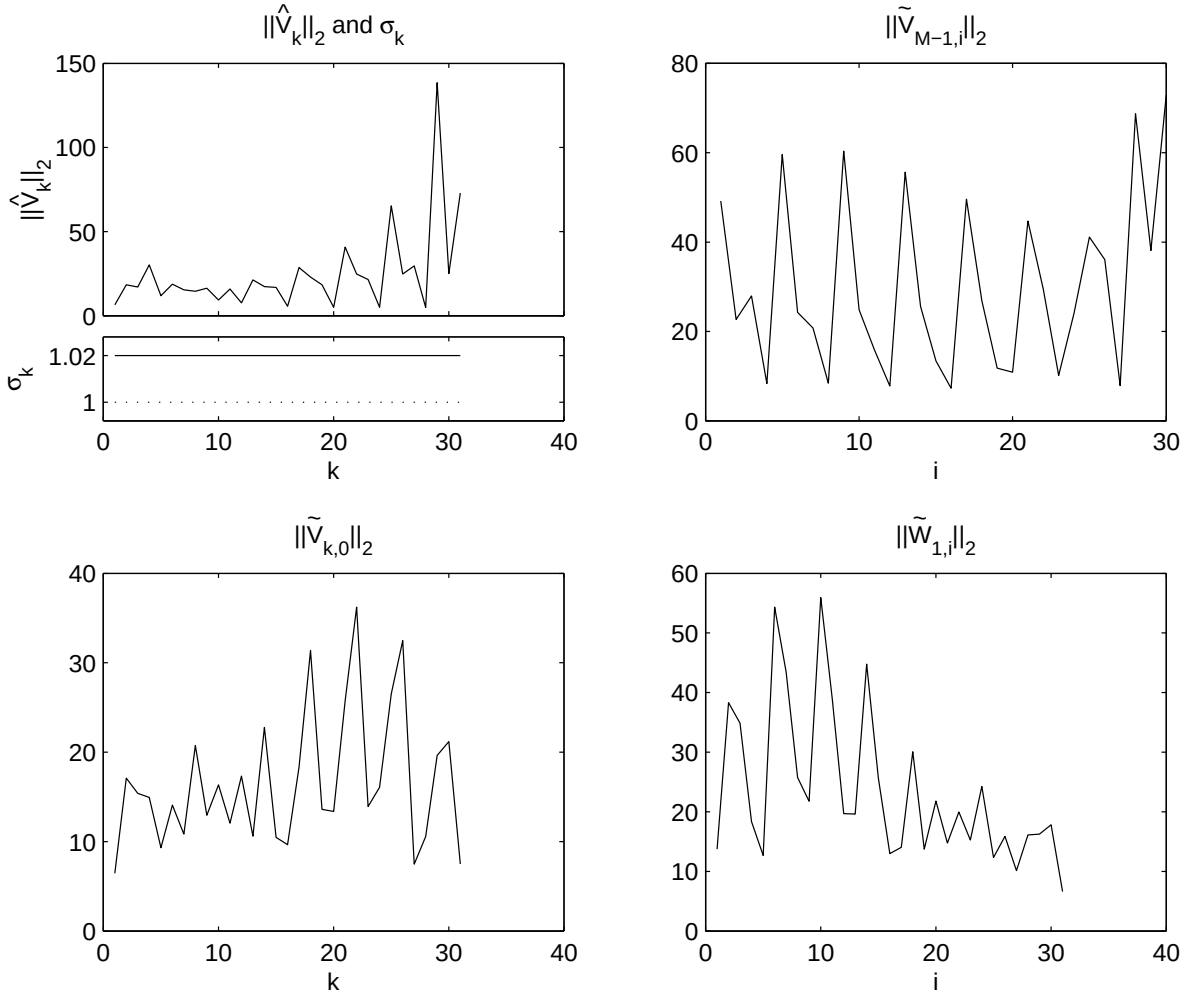


Figure A.8: The σ_k-RSCALE solution ($\tau_{\hat{V}} = 10.0,\ \epsilon_\sigma = 0.25$) to Table 3.1/D.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
10, ?	?	?	?	0.0039	0.0023	3.9e+04	3.1e-11

The true value of $\mathcal{K}_2(C)$ is 1.1e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.89) (bottom right):

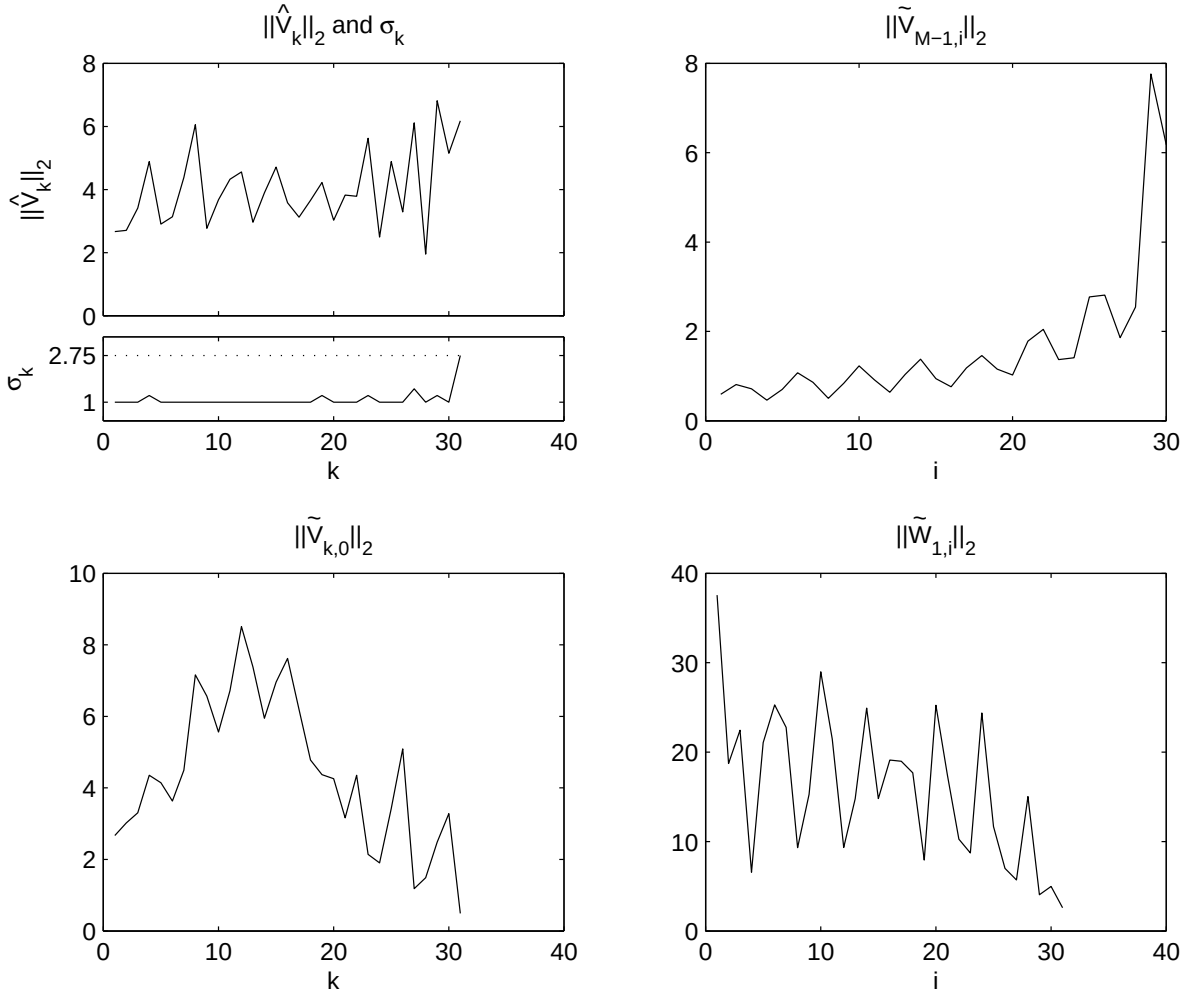


Figure A.9: The static 1.0-RSCALE solution to Table 3.1/E.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\|\tilde{V}_{k,i}\|\|_2$	$\|\|\tilde{W}_{k,i}\|\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 4.1e-15	1.4e+16	1.4e+16	6.9e+30	0.00024	1.8e+11	5.4e+04	1.8e+11

The true value of $\mathcal{K}_2(C)$ is 30. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

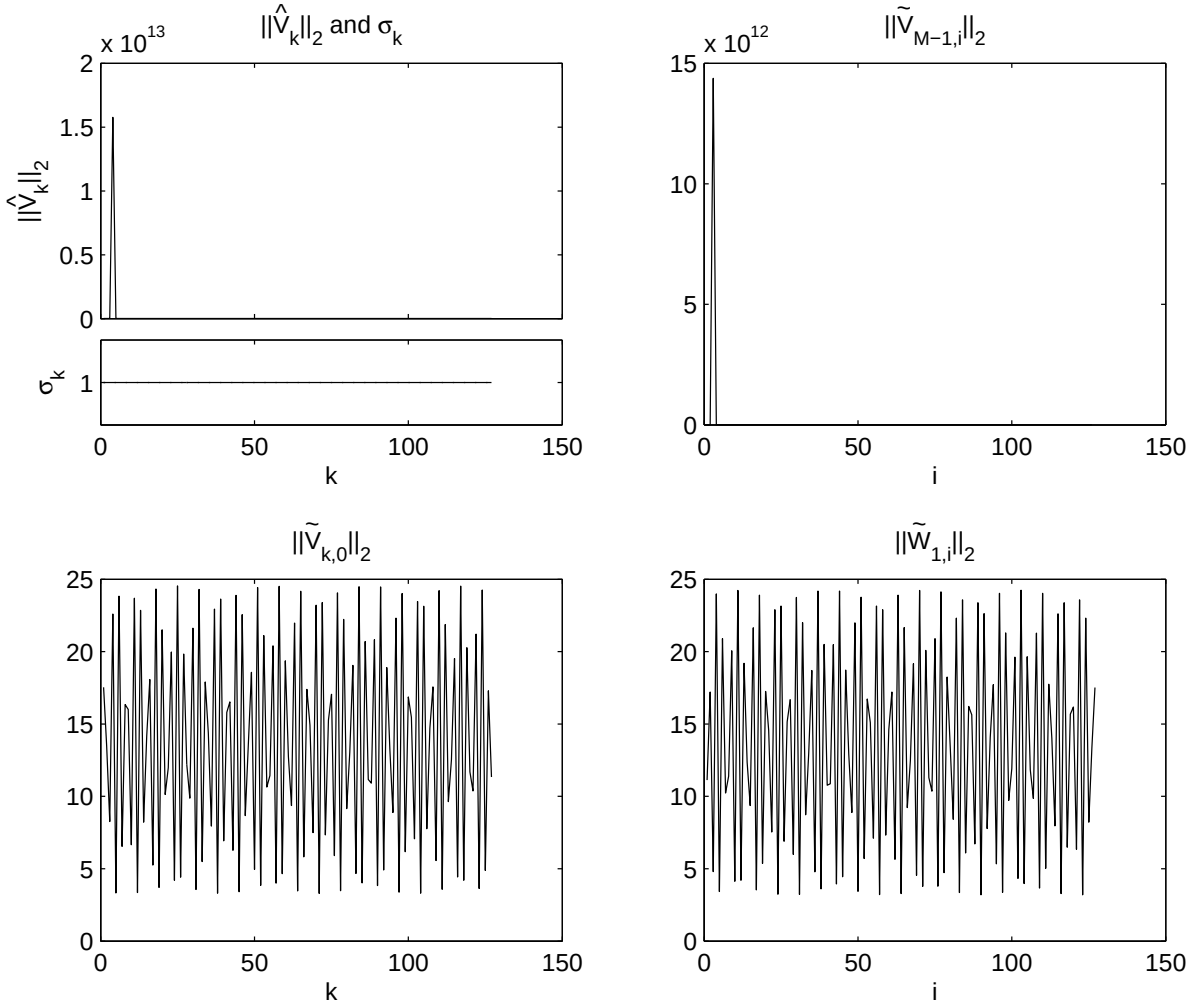


Figure A.10: The static 0.98-RSCALE solution to Table 3.1/E.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
ω_σ, ω_λ	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
13, 0.02	3.9e+04	2.9e+03	5.1e+07	0.00024	0.00017	5.4e+04	1.8e-10

The true value of $\mathcal{K}_2(C)$ is 1.8e+02. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

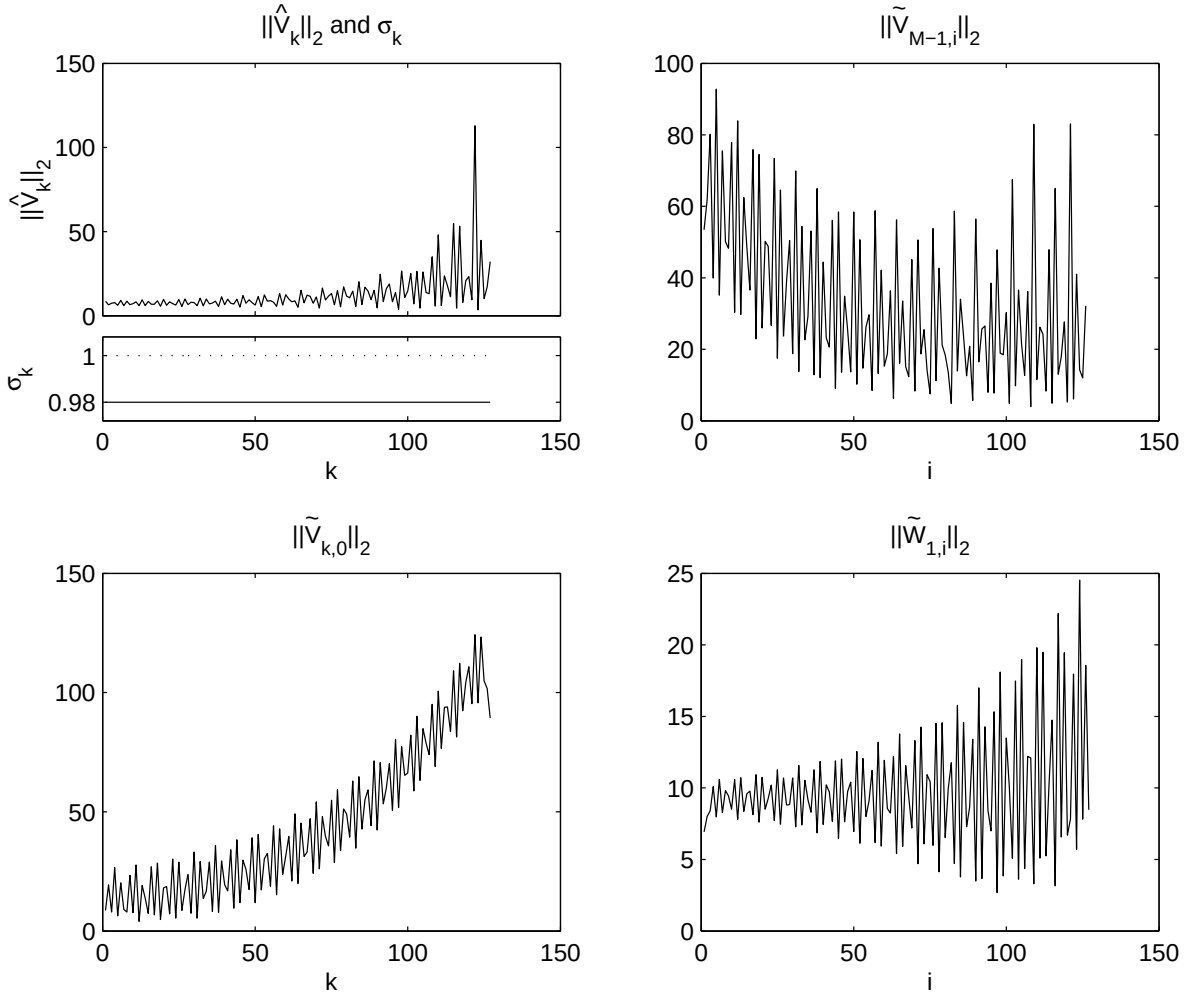


Figure A.11: The static 1.02-RSCALE solution to Table 3.1/E.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
13, 0.02	2.9e+03	2.9e+03	4.6e+07	0.00024	0.00017	5.4e+04	1.9e-10

The true value of $\mathcal{K}_2(C)$ is 78. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

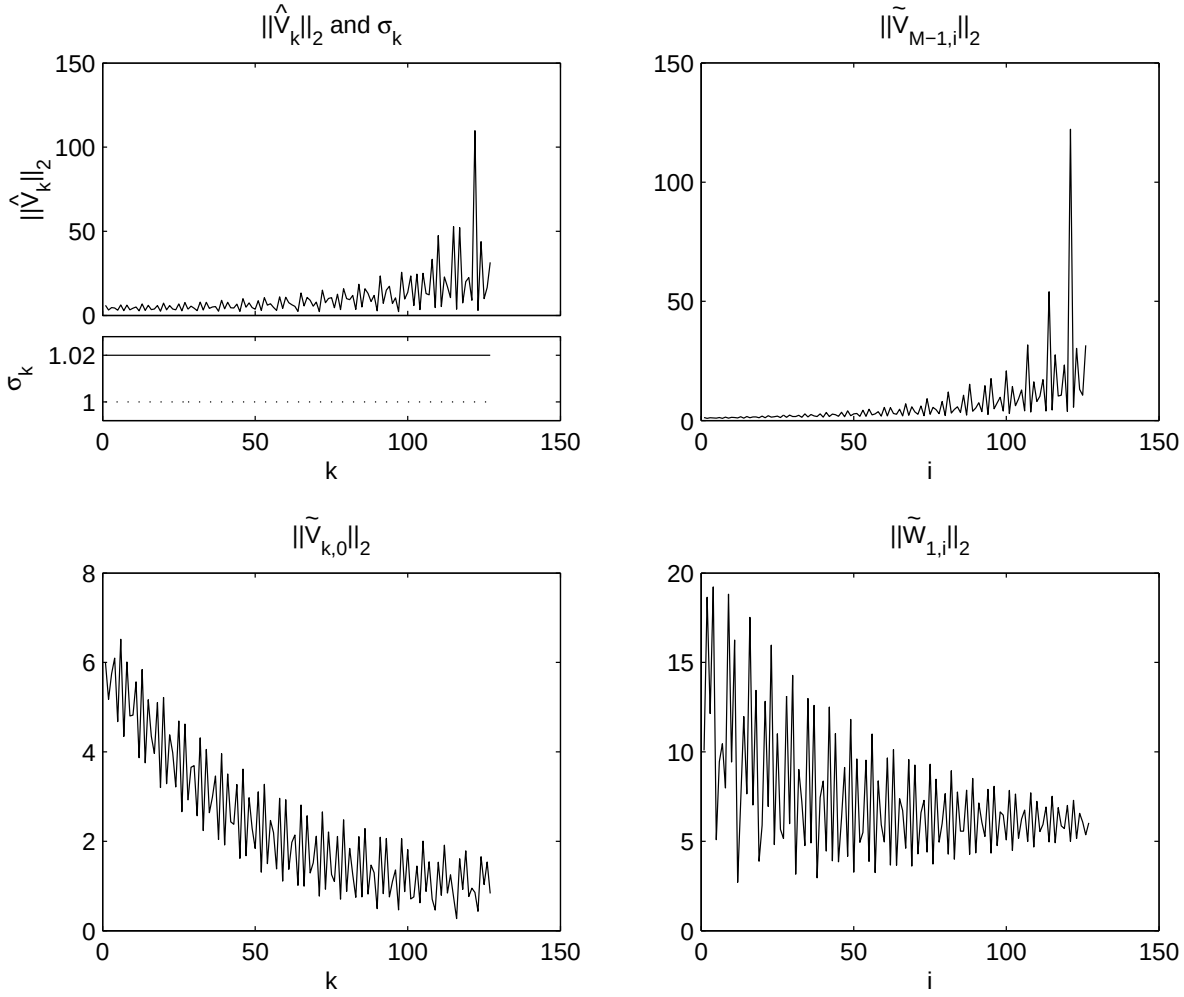


Figure A.12: The σ_k-RSCALE solution ($\tau_{\hat{V}} = 10.0,\ \epsilon_\sigma = 0.25$) to Table 3.1/E.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
8.8, ?	?	?	?	0.00024	0.00017	5.4e+04	4.7e-11

The true value of $\mathcal{K}_2(C)$ is 66. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.89) (bottom right):

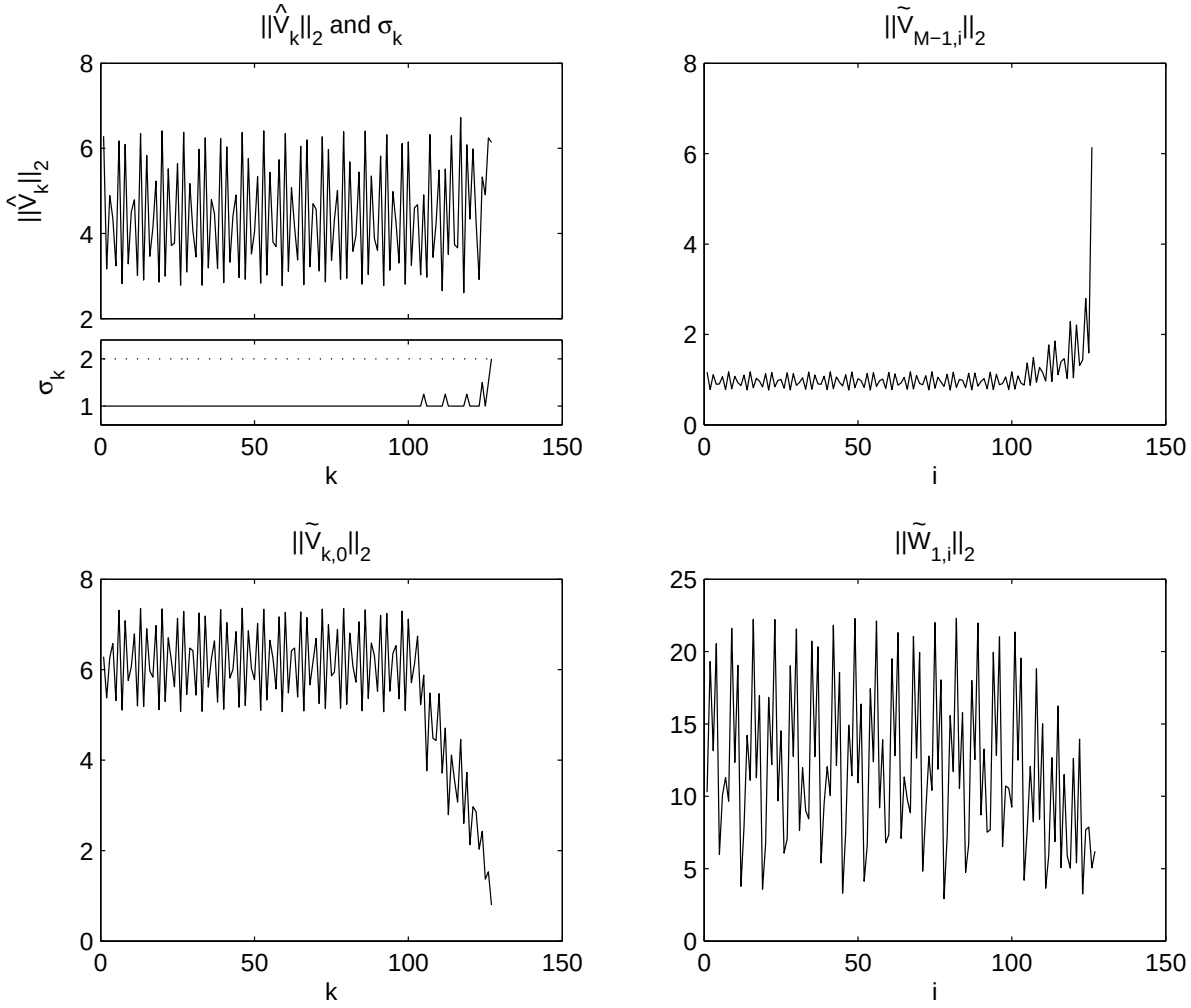


Figure A.13: The static 1.0-RSCALE solution to Table 3.1/F.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\|\tilde{V}_{k,i}\|\|_2$	$\|\|\tilde{W}_{k,i}\|\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
1, 6.8e-15	1e+16	1e+16	3e+30	0.00098	72	5.8e+04	72

The true value of $\mathcal{K}_2(C)$ is 5.4e+13. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

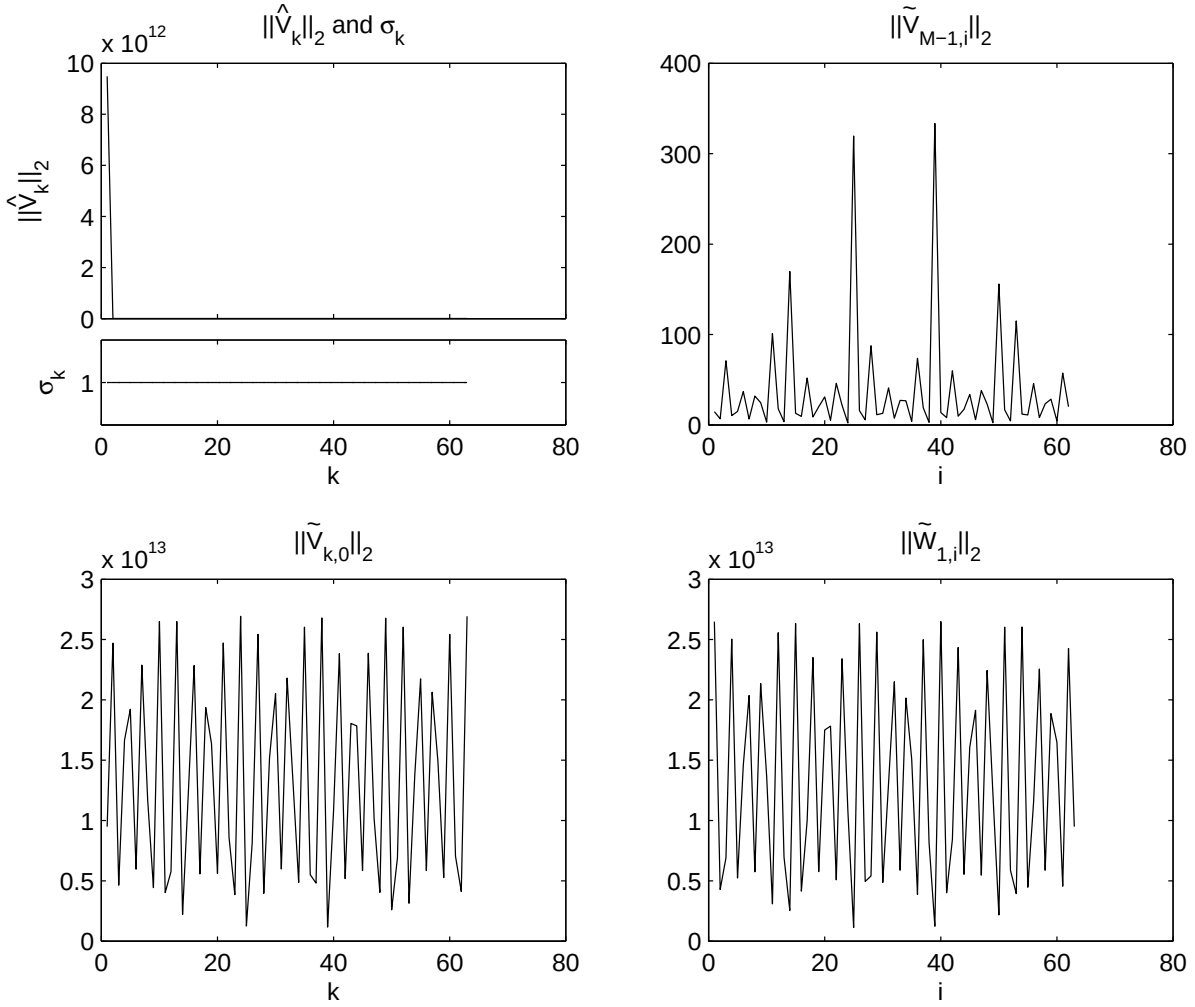


Figure A.14: The static 0.98-RSCALE solution to Table 3.1/F.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.6, 0.02	1.3e+04	3.4e+03	4.5e+06	0.00098	0.00038	5.8e+04	2.9e-11

The true value of $\mathcal{K}_2(C)$ is 75. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

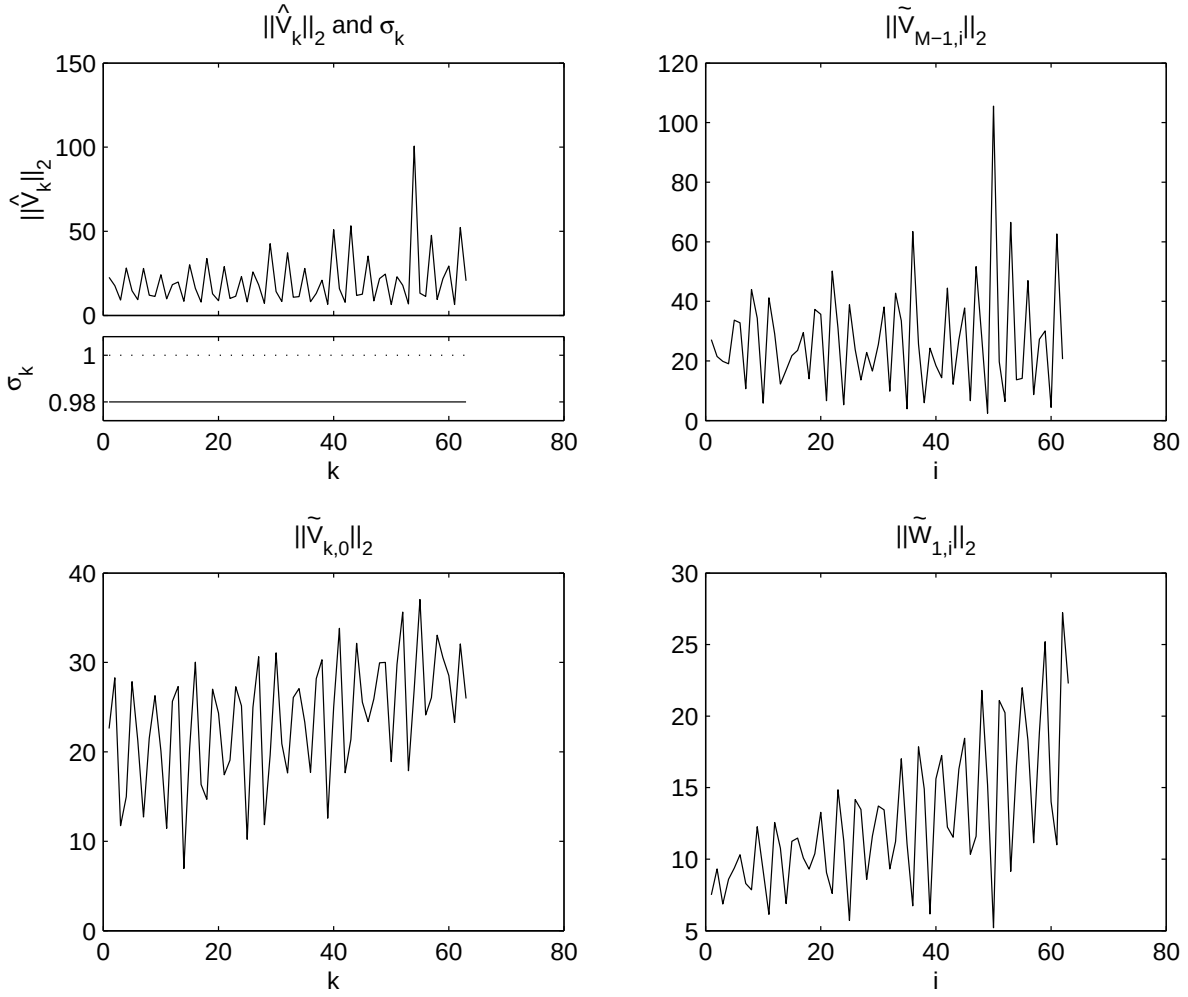


Figure A.15: The static 1.02-RSCALE solution to Table 3.1/F.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
3.6, 0.02	3.4e+03	3.4e+03	4.3e+06	0.00098	0.00038	5.8e+04	2.7e-11

The true value of $\mathcal{K}_2(C)$ is 48. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.12) (bottom right):

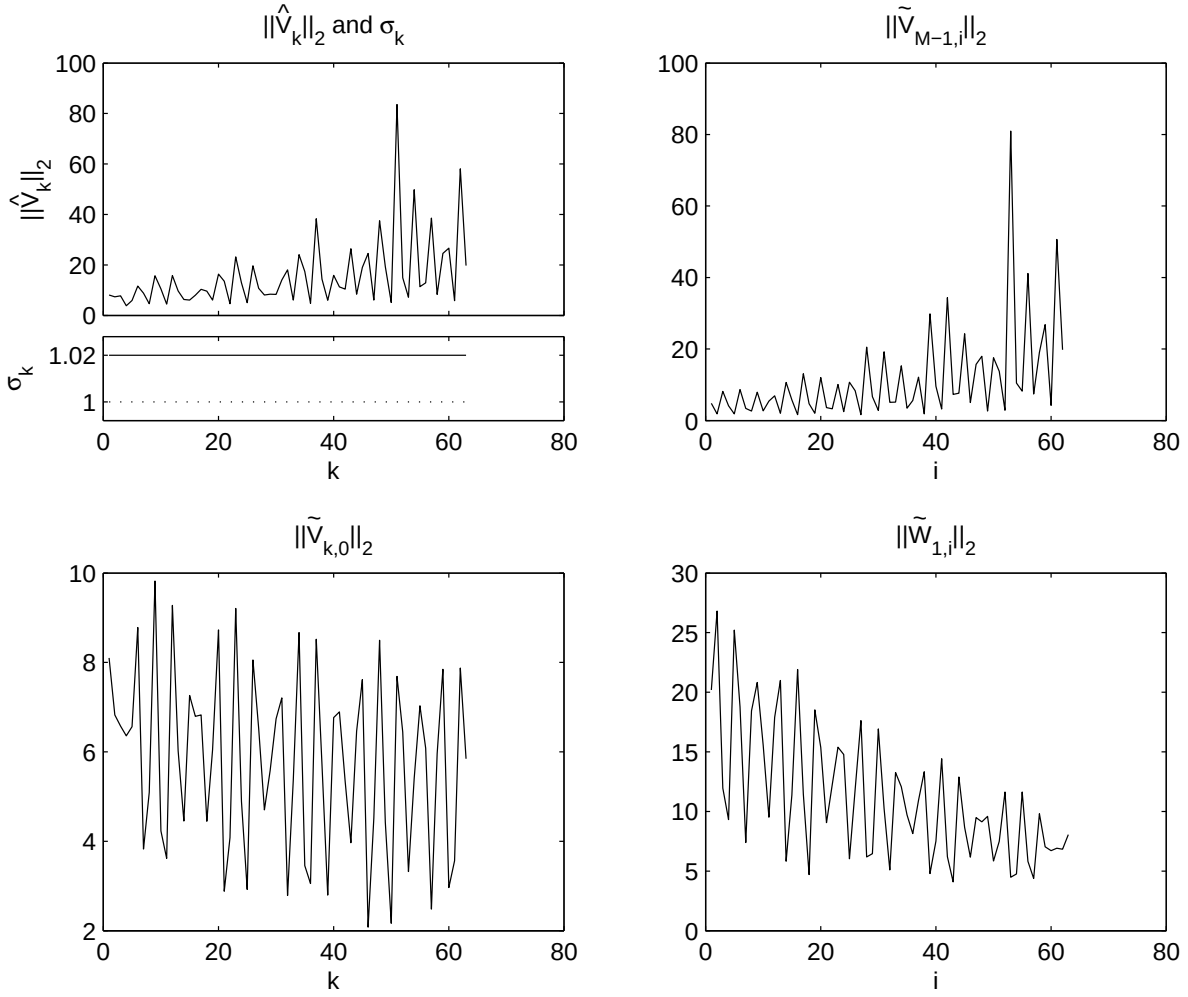
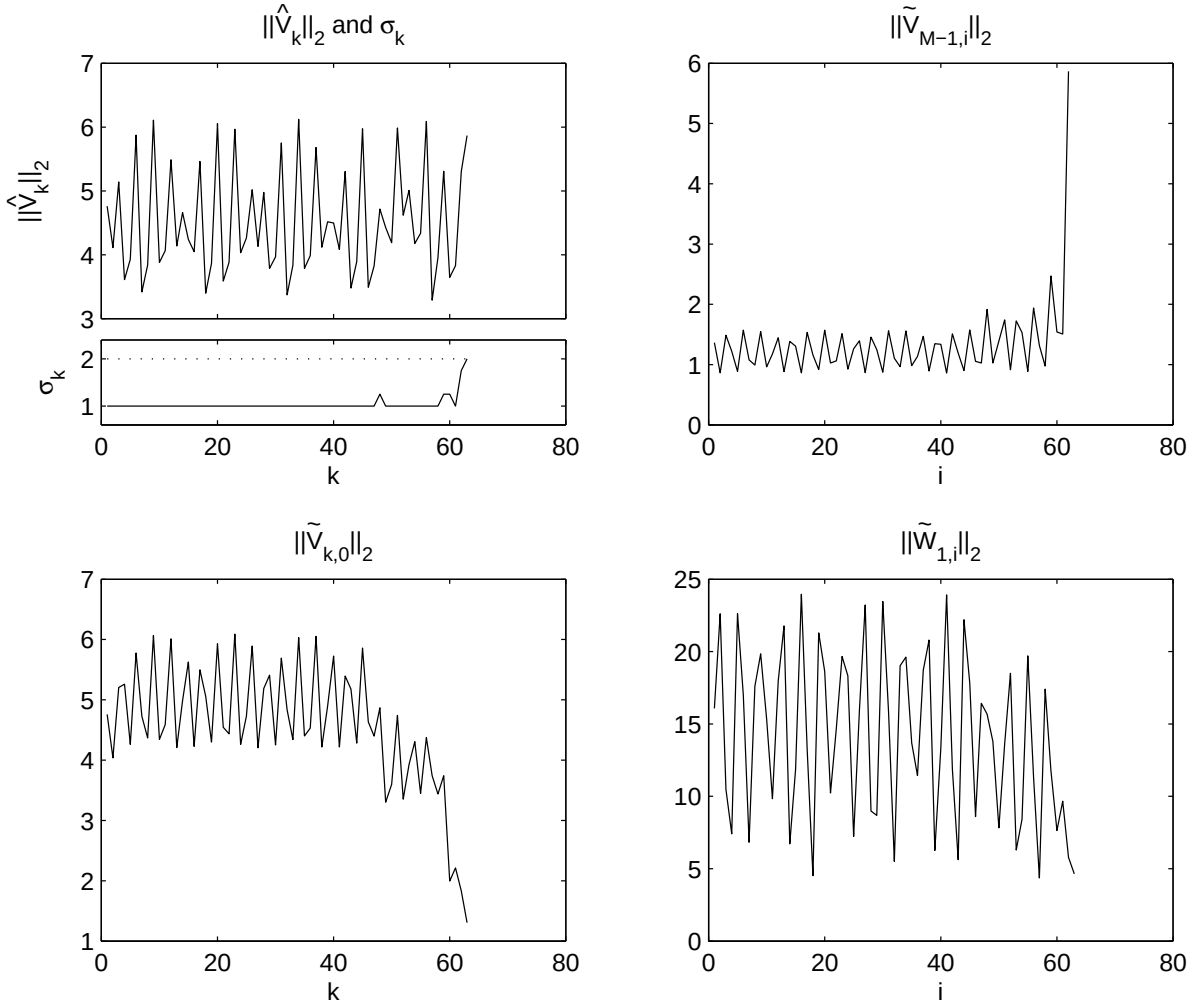


Figure A.16: The σ_k-RSCALE solution ($\tau_{\hat{V}} = 10.0,\ \epsilon_\sigma = 0.25$) to Table 3.1/F.

Theoretical bounds.				Accuracy of $\bar{Y}$.			
$\omega_\sigma,\ \omega_\lambda$	$\|\tilde{V}_{k,i}\|_2$	$\|\tilde{W}_{k,i}\|_2$	$\mathcal{K}_2(C)$	h^2	ana. err.	$\bar{\mathcal{K}}_1(\mathcal{J})$	alg. err.
6.8, ?	?	?	?	0.00098	0.00038	5.8e+04	2.5e-11

The true value of $\mathcal{K}_2(C)$ is 45. The following plots show the true $\|\hat{V}_k\|_2$ along with the eigenvalue shifts used during rescaling (top left), sample norms arising in the computation of $\tilde{\phi}_{M-1}$ in (3.14) (top right), $\tilde{y}_k$ in (3.16) (bottom left), and $\hat{\phi}_1$ in (3.89) (bottom right):



Appendix B

Additional Sequential Experiments

This appendix contains the results of additional numerical experiments from §3.2.2 (Figures B.1-B.4) and §4.2 (Figures B.5-B.10).

Figures B.1-B.4 show the results of experiments where we generate hundreds of linear problems in search of problems that cause difficulty for *SLF*-LU. We divide the test DEs into three classes, based on the sign of the elements in the Jacobian. Each problem is discretized using trapezoidal finite differences, the resulting ABD system is reduced with *SLF*-LU, and block growth is monitored during the reduction in order to detect potential *SLF*-LU instability. (See §3.2.2 for further details on the criteria used to detect instability.) The effect of Jacobian order and sparsity on stability is investigated in Figures B.1 and B.2, respectively. The effect of Jacobian scale with dense Jacobians is investigated in Figure B.3, and the effect of Jacobian scale with sparser ($\rho = 50\%$ nonzero) Jacobians is investigated in Figure B.4.

Figures B.5-B.10 show the results of additional numerical experiments illustrating the relative performance of the three ABD system solvers *SLF*-QR, *SLF*-LU, and RSCALE, when run in sequential mode, on a sequential machine. Experiments on problems generated from classes K, L and M of Table 4.3 in §4.2 are included. In each of these problems, the Jacobian of the DE is random in structure—the nonzeros of the Jacobian are randomly-generated and randomly-distributed throughout the matrix, with at least one nonzero in each row and column. Experimental results are shown for sixteen Class K problems ($n = 10$) in Figures B.5 and B.6, sixteen Class L problems ($n = 12$) in Figures B.7 and B.8, and sixteen Class M problems ($n = 14$) in Figures B.9 and B.10. In most experiments, as Jacobian density is increased from 20% to 90% nonzero, we see a trend toward increased *SLF*-LU execution time. Note that *SLF*-LU exhibits instability when solving some problems in Figures B.6, B.7, B.8 and B.9.

Figure B.1: Effect of Jacobian order (n) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 dense linear problems with $\lambda = 100$ and $\rho = 100\%$ nonzero, on meshes ranging from $h = 0.02$ to $h = 0.09$.

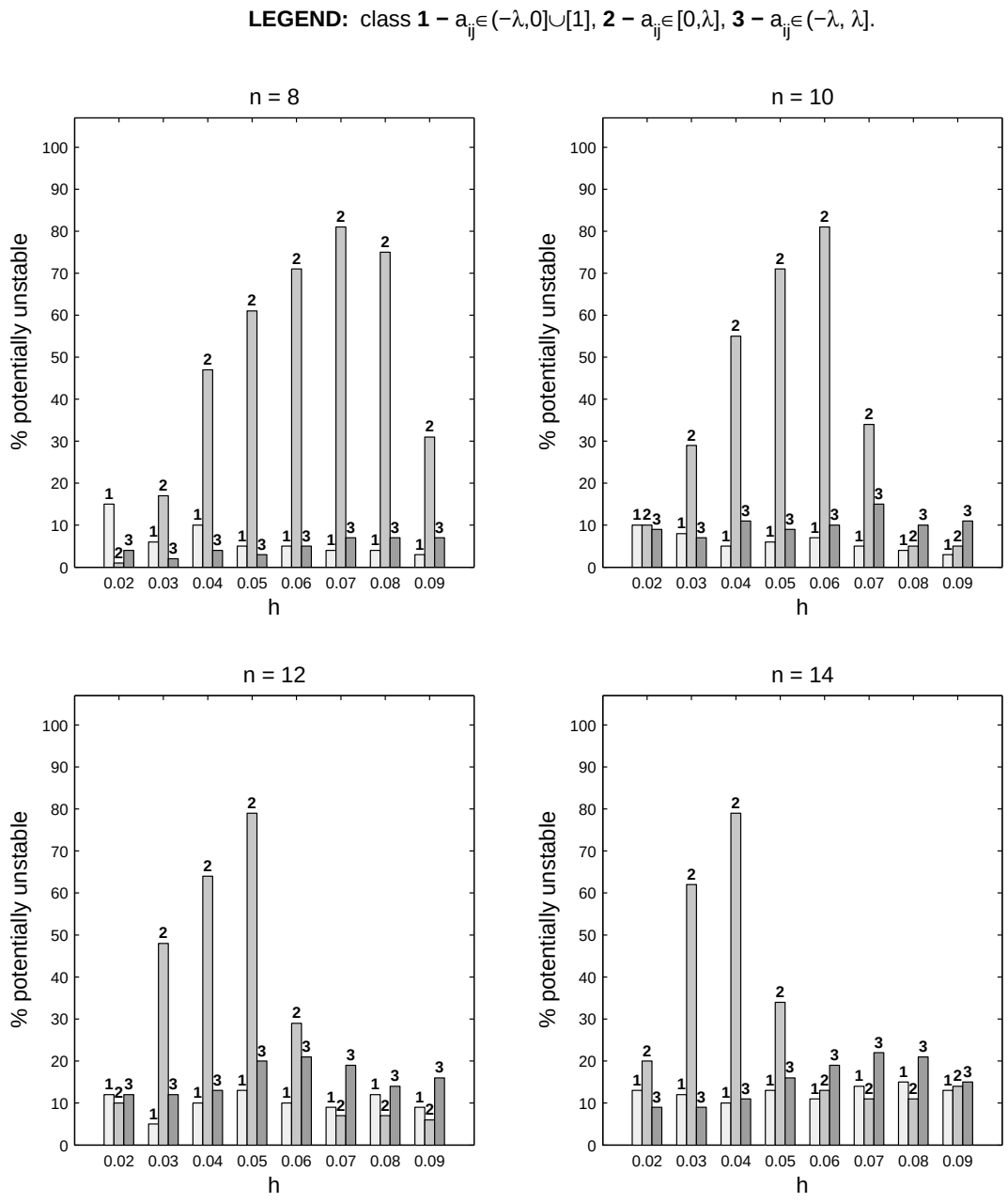


Figure B.2: Effect of Jacobian sparsity (ρ) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 unstructured linear problems with $\lambda = 100$ and $n = 10$, on meshes ranging from $h = 0.02$ to $h = 0.09$. Jacobian nonzeros are randomly distributed.

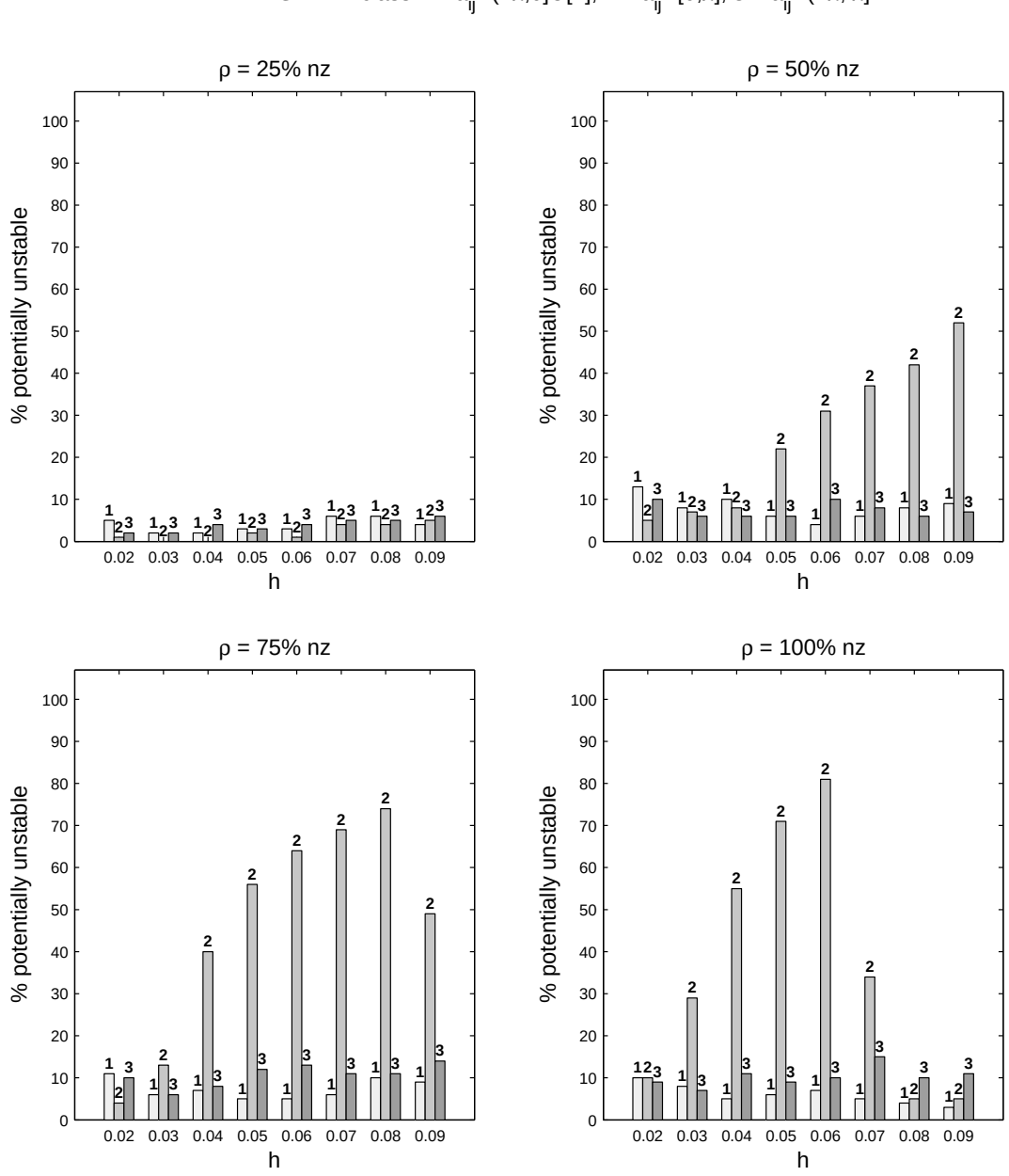


Figure B.3: Effect of Jacobian scale (λ) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 dense linear problems with $n = 10$ and $\rho = 100\%$ nonzero, on meshes ranging from $h = 0.2$ to $h = 0.9$.

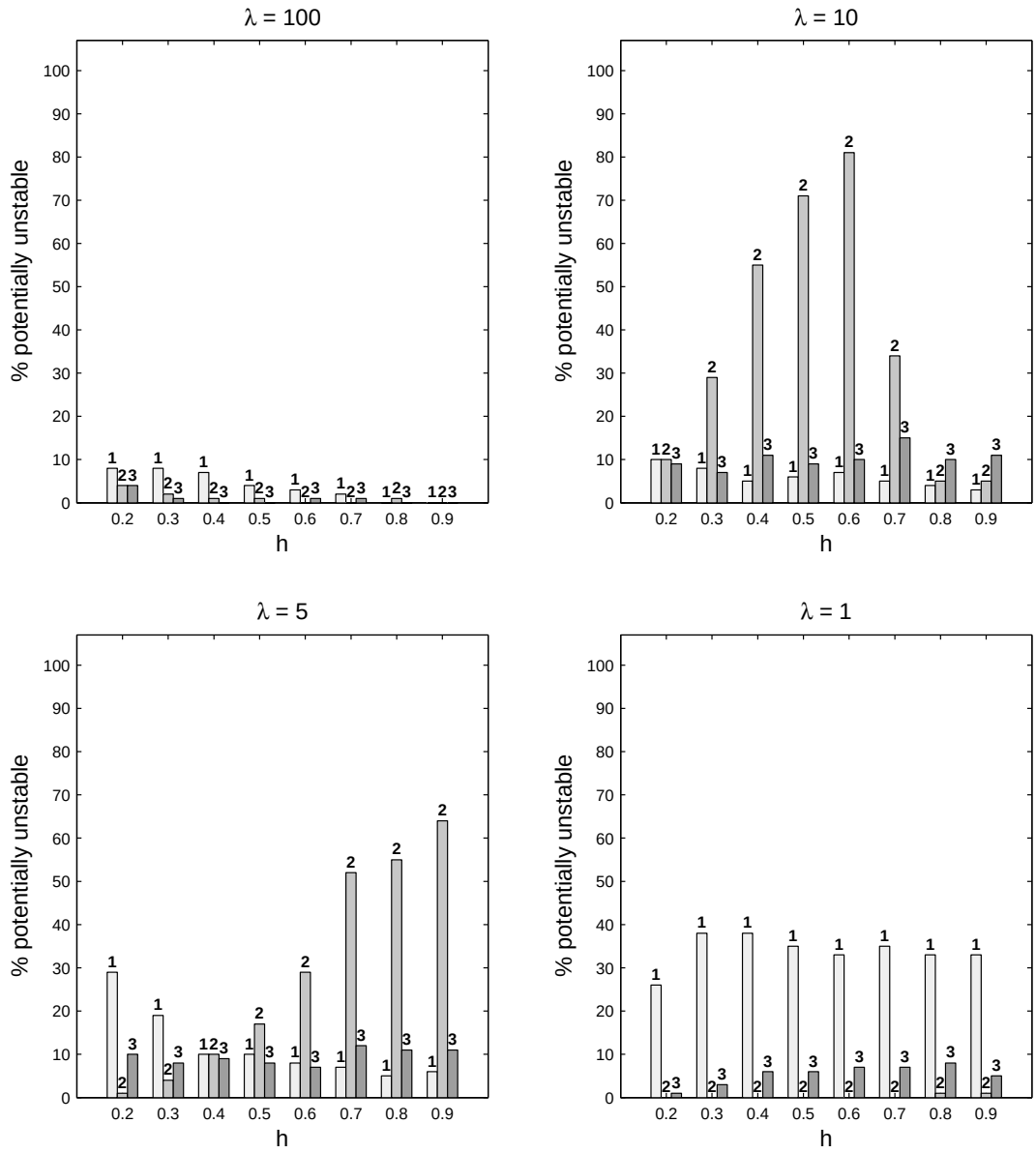


Figure B.4: Effect of Jacobian scale (λ) on *SLF*-LU stability when solving 100 randomly-generated class 1, 2 and 3 unstructured linear problems with $n = 10$ and $\rho = 50\%$ nonzero, on meshes ranging from $h = 0.2$ to $h = 0.9$. Jacobian nonzeros are randomly distributed.

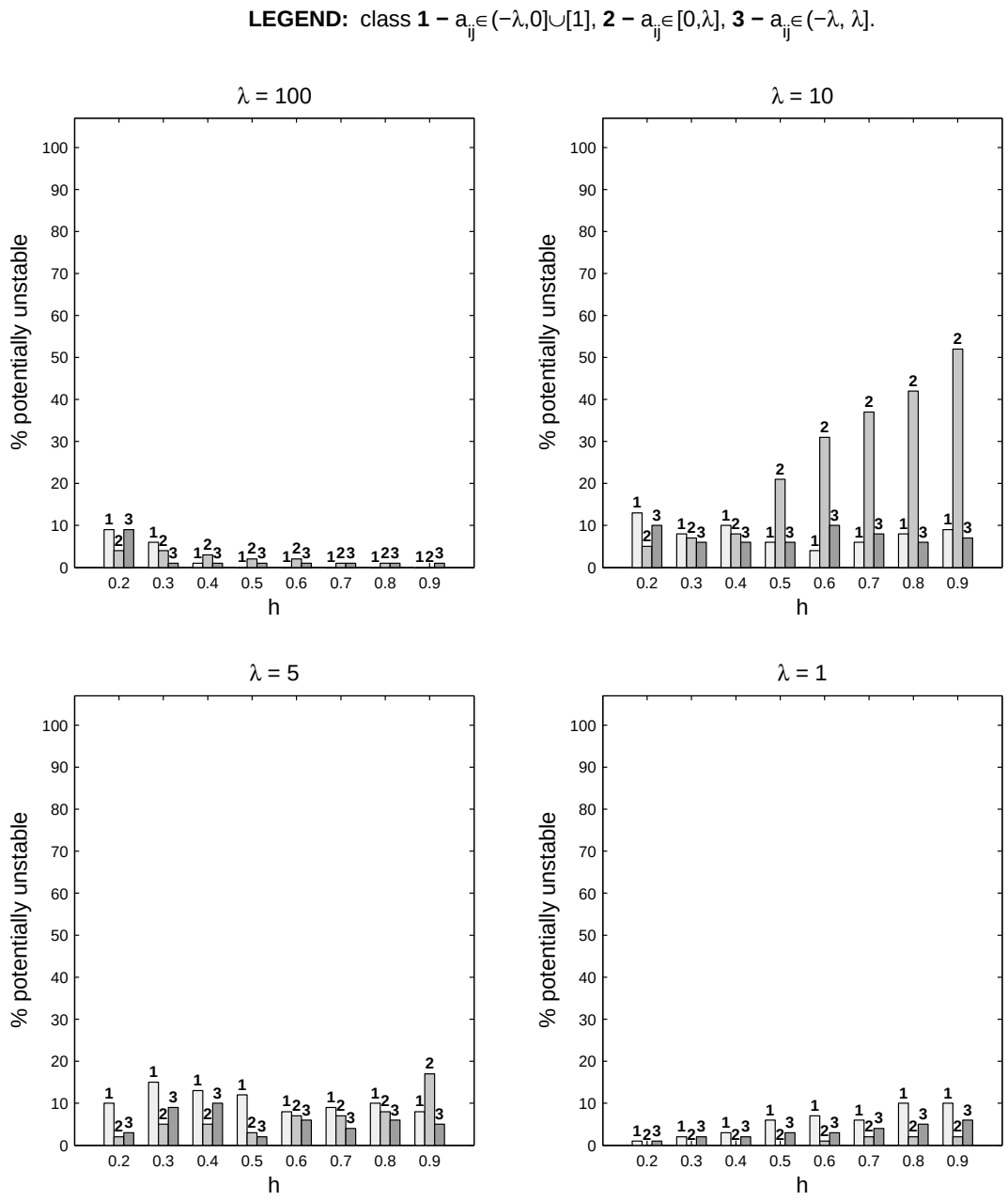


Figure B.5: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class K problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.

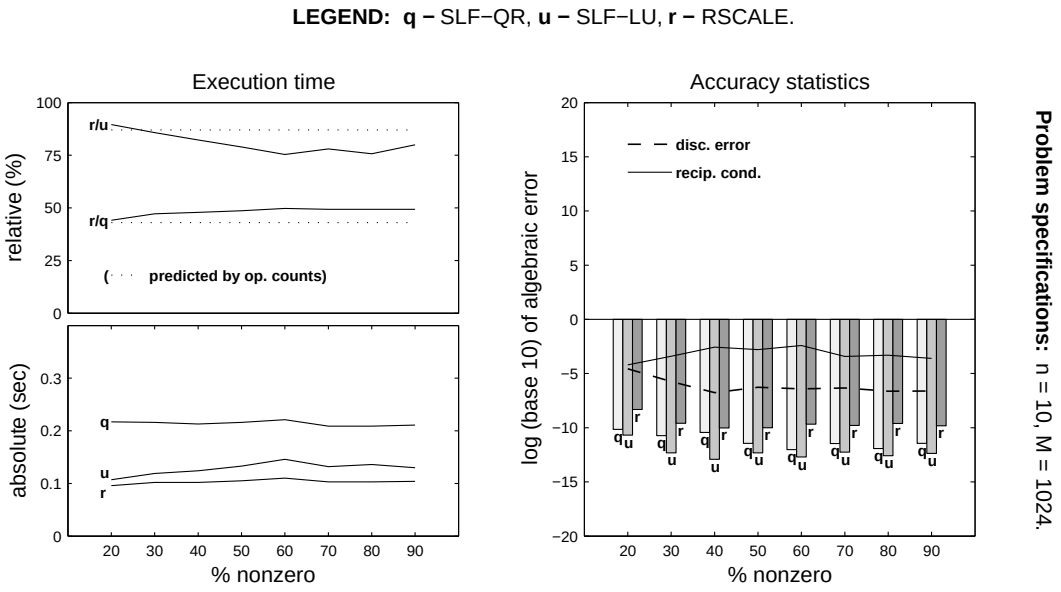


Figure B.6: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class K problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.

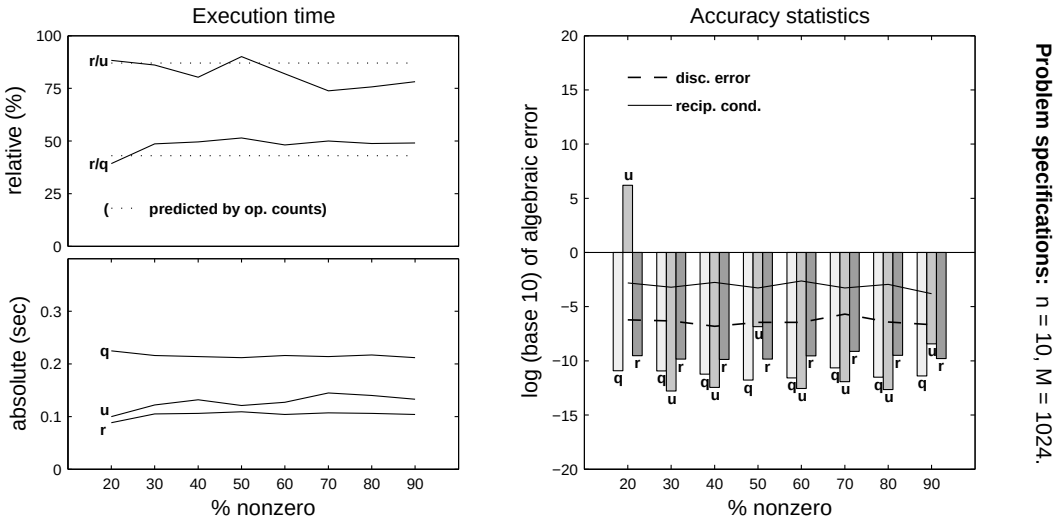


Figure B.7: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class L problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.

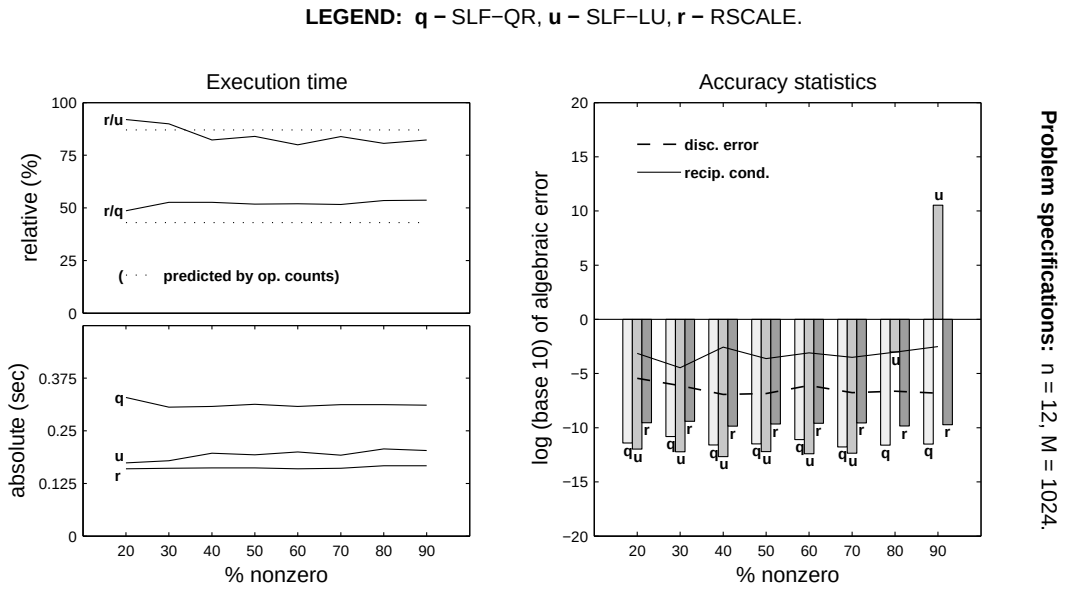


Figure B.8: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class L problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.

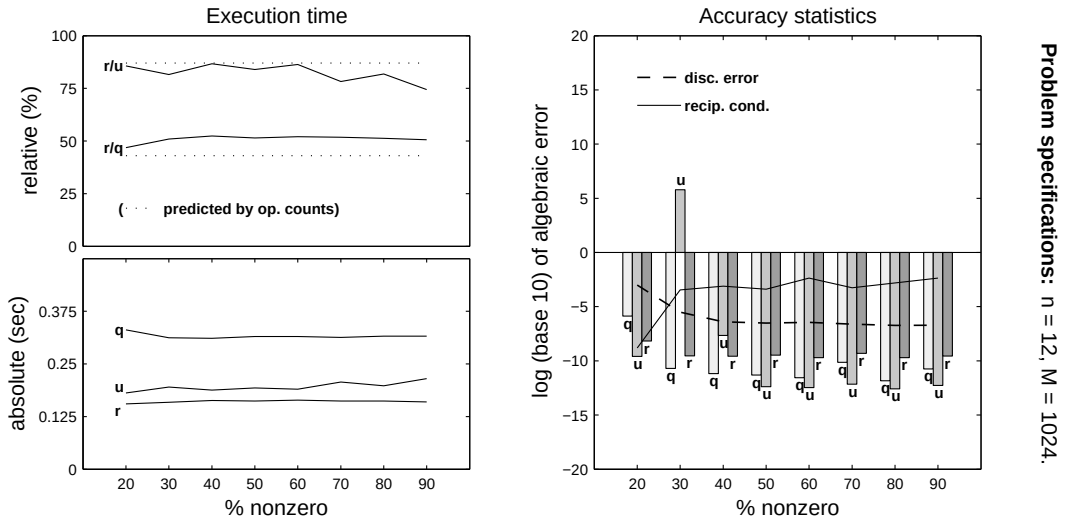


Figure B.9: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class M problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.

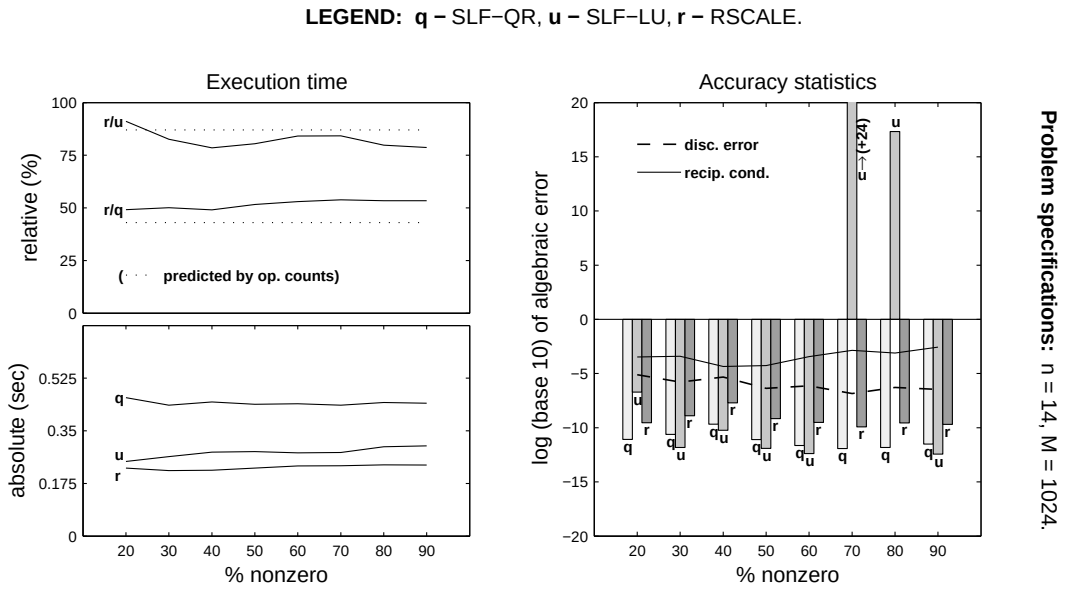
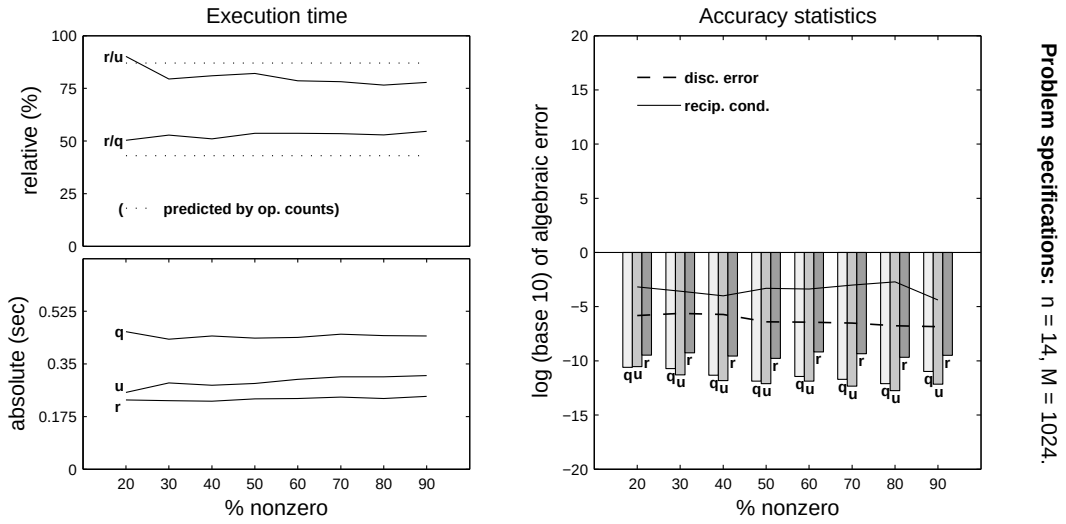


Figure B.10: Execution time and accuracy of the three sequential ABD solvers when solving eight randomly-generated Class M problems of Table 4.3. Jacobian sparsity varies from 20% to 90% nonzero, with nonzeros randomly distributed.



Appendix C

Additional Parallel Experiments

This appendix contains the results of additional numerical experiments illustrating the relative performance of the three ABD system solvers *SLF*-QR, *SLF*-LU, and RSCALE, when run in parallel mode, on a parallel machine. Additional experiments on problems A-F of Table 4.5 in §4.3 are included.

Figures C.1-C.6 show results for problems A-F of Table 4.5 with $M = 1024$. Figures C.7-C.12 show results with $M = 2048$. In both sets of experiments, as Jacobian order increases from $n = 6$ to $n = 16$ the payoff of parallelism as defined in §4.3 occurs with fewer processors, and speedups become more optimal. Payoff for *SLF*-QR occurs only in Figures C.11 and C.12 ($M = 2048$, $n = 14$ and $n = 16$).

Figure C.1: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem A of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 7 processors; there is no payoff when using *SLF*-QR.

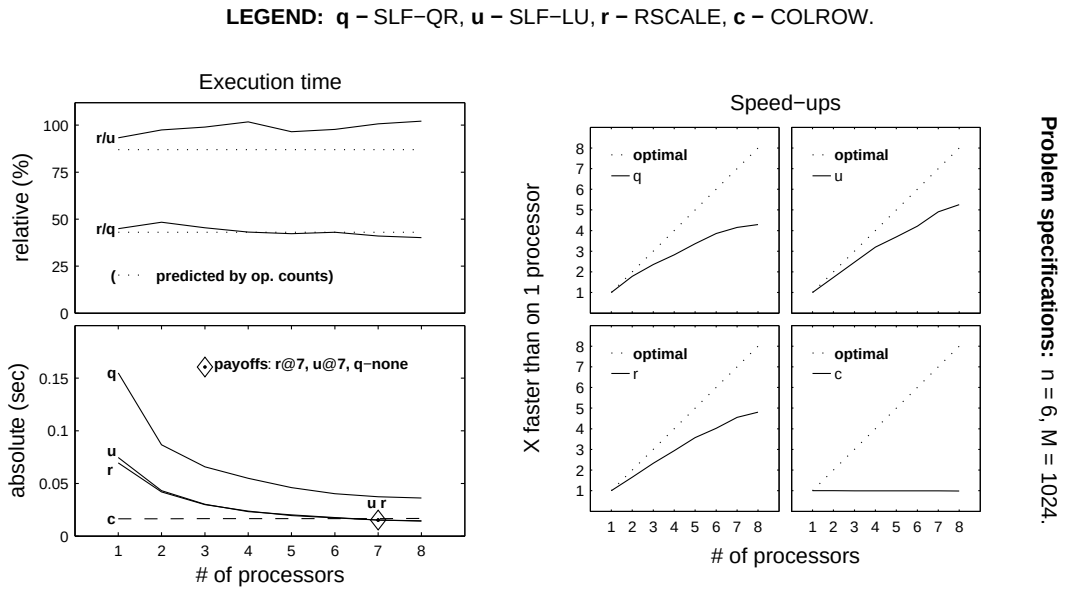


Figure C.2: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem B of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 6 processors; there is no payoff when using *SLF*-QR.

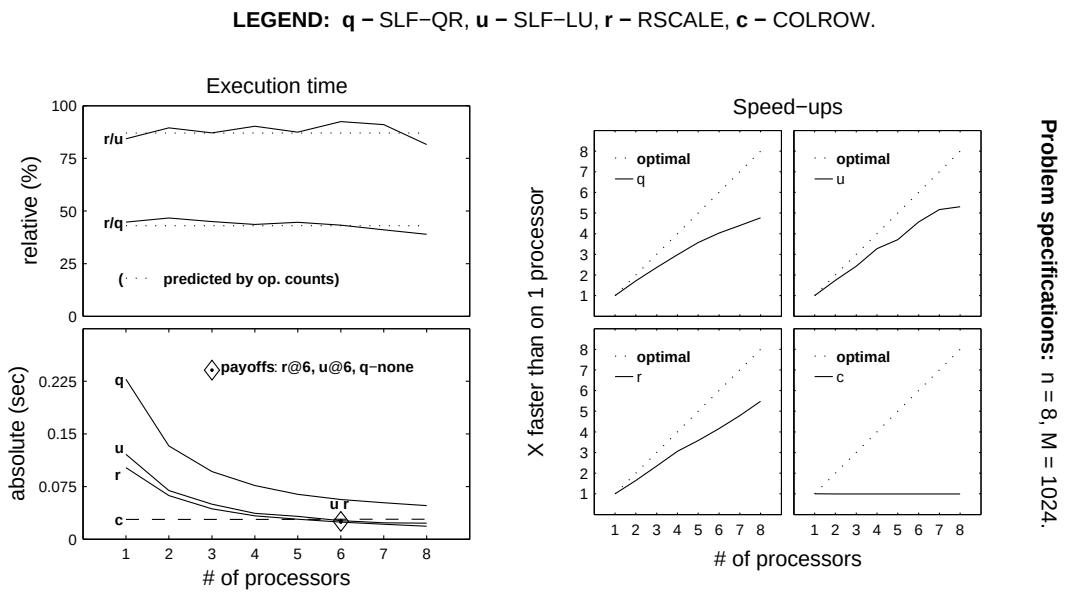


Figure C.3: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem C of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 5 processors; there is no payoff when using *SLF*-QR.

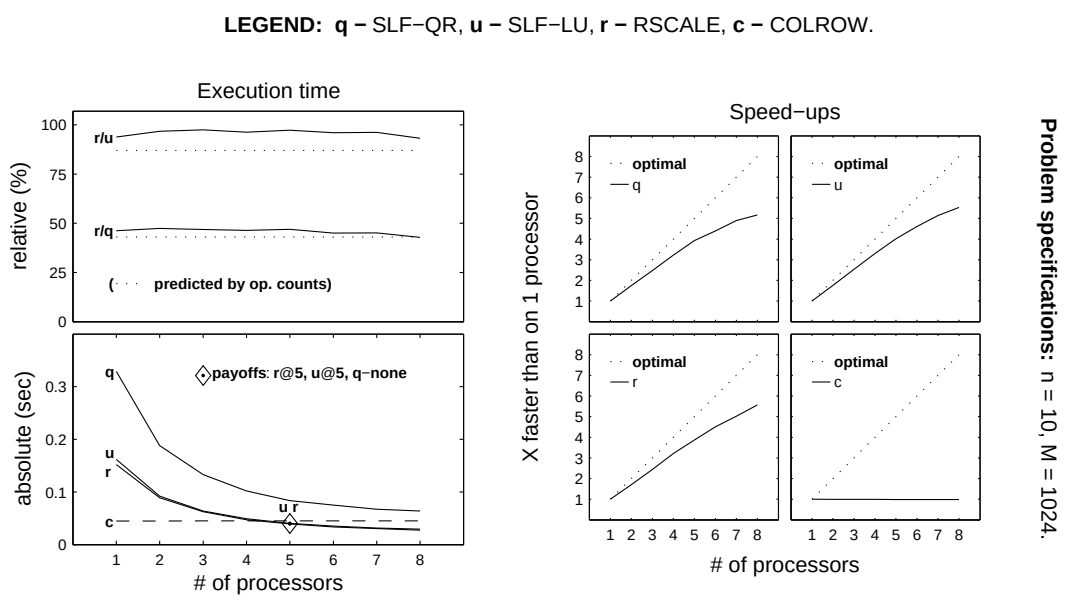


Figure C.4: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem D of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU payoff at 4 and 5 processors, respectively; there is no payoff when using *SLF*-QR.

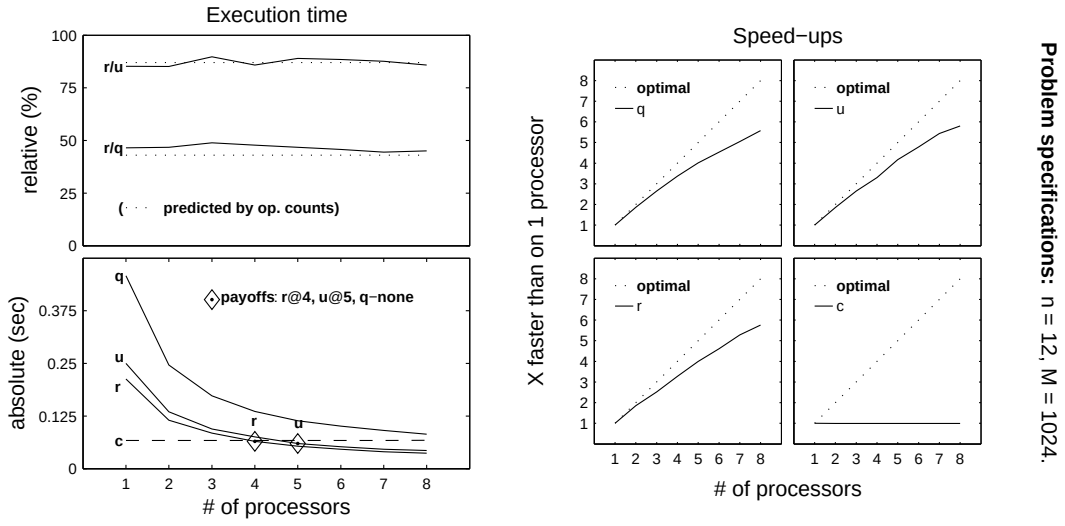


Figure C.5: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem E of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU payoff at 4 and 5 processors, respectively; there is no payoff when using *SLF*-QR.

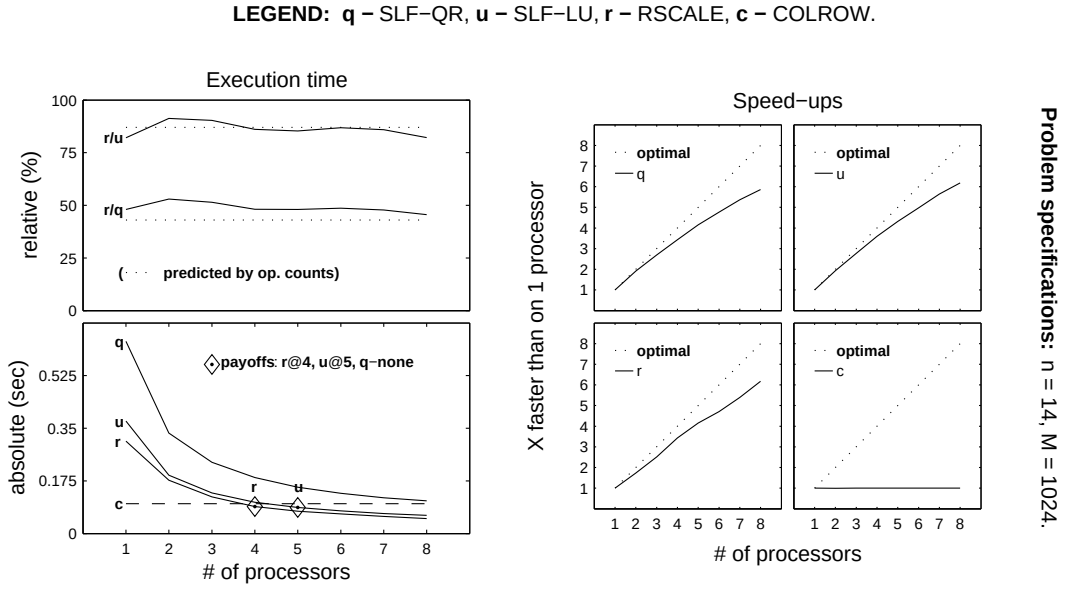


Figure C.6: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem F of Table 4.5, with $M = 1024$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 4 processors; there is no payoff when using *SLF*-QR.

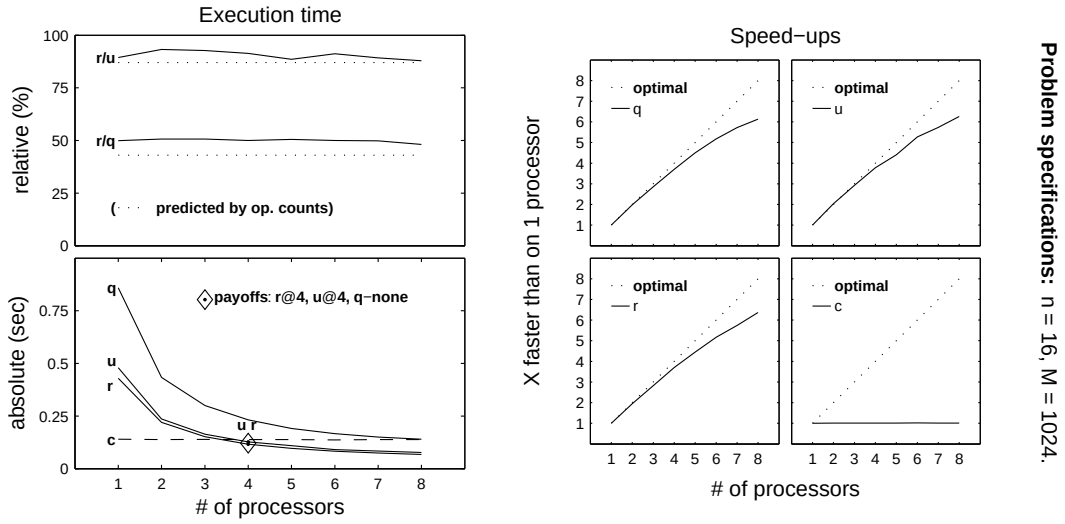


Figure C.7: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem A of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 7 processors; there is no payoff when using *SLF*-QR.

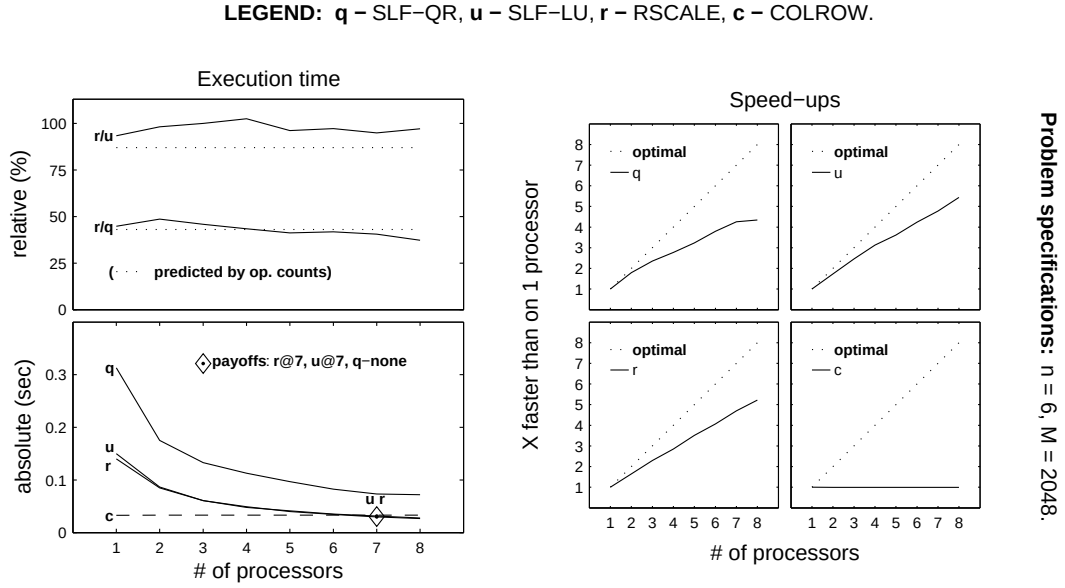


Figure C.8: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem B of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU payoff at 5 and 6 processors, respectively; there is no payoff when using *SLF*-QR.

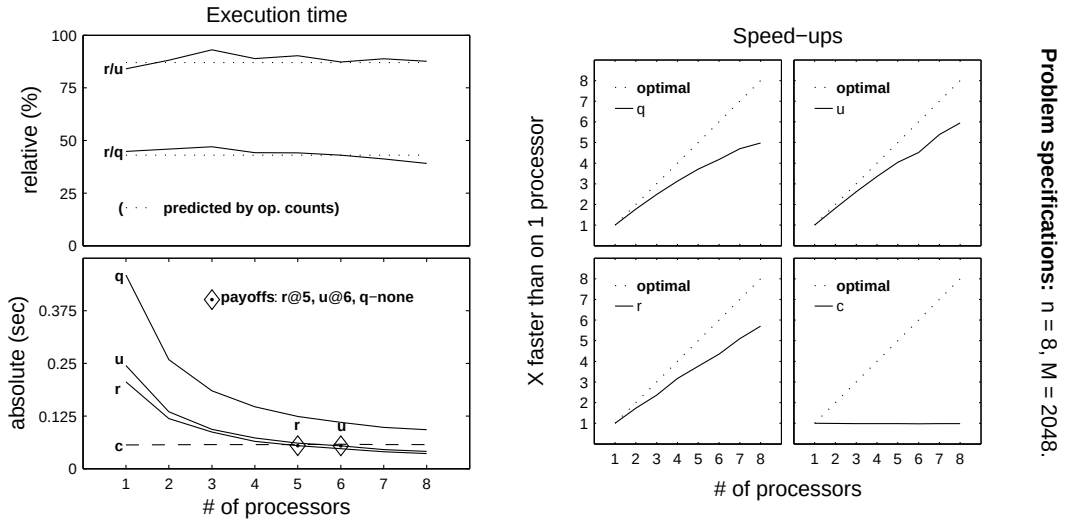


Figure C.9: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem C of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU each payoff at 5 processors; there is no payoff when using *SLF*-QR.

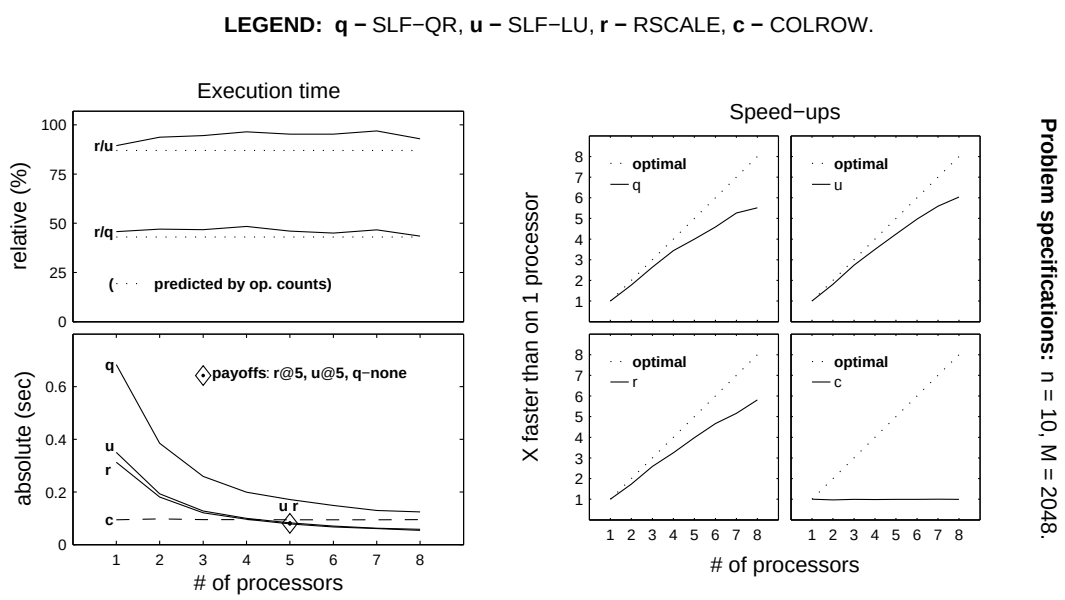


Figure C.10: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem D of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE and *SLF*-LU payoff at 4 and 5 processors, respectively; there is no payoff when using *SLF*-QR.

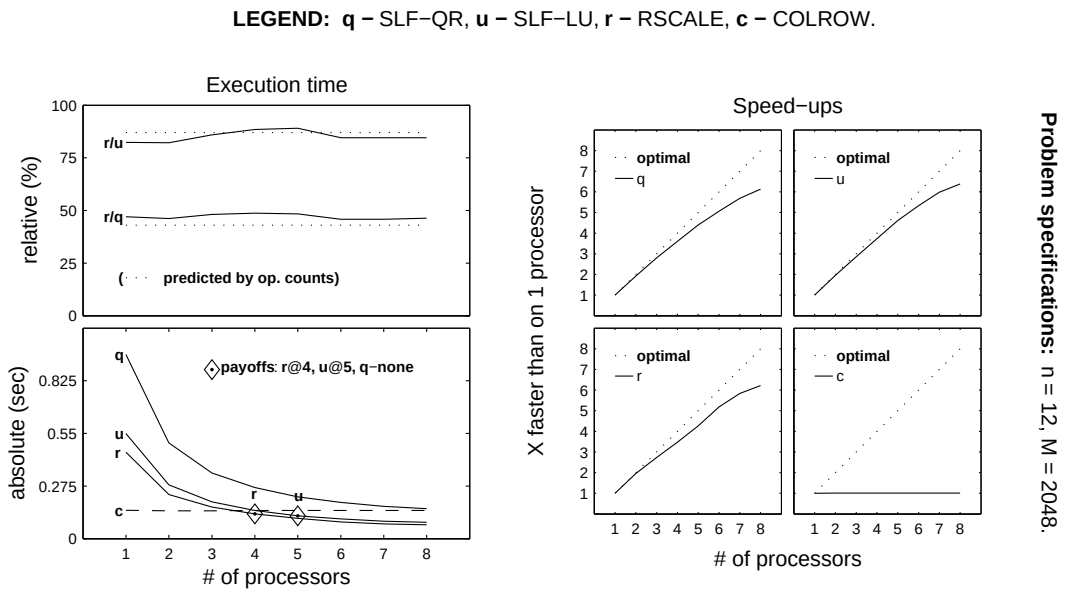


Figure C.11: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem E of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE, *SLF*-LU and *SLF*-QR payoff at 4, 4 and 7 processors, respectively.

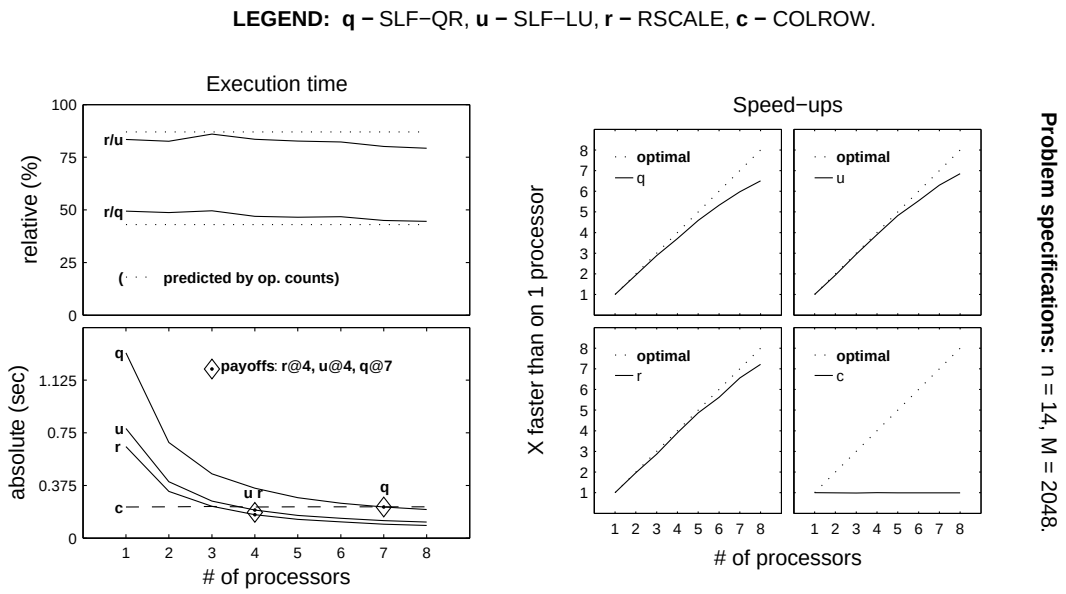
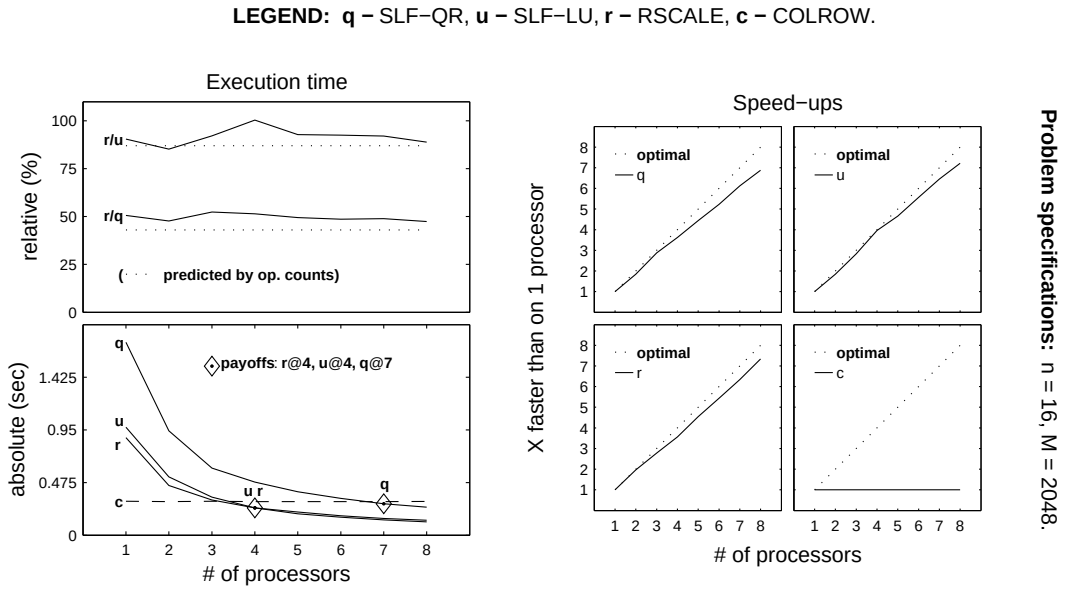


Figure C.12: Execution time and speed-ups of the three parallel ABD solvers and COLROW when solving Problem F of Table 4.5, with $M = 2048$. Architecture is the SGI Origin 2000. RSCALE, *SLF*-LU and *SLF*-QR payoff at 4, 4 and 7 processors, respectively.



Appendix D

Additional MirkDC Performance Experiments

This appendix contains the results of several additional numerical experiments illustrating the sequential and parallel performance of the four variants of MirkDC—MirkDC/COLROW, MirkDC/*SLF*-QR, MirkDC/*SLF*-LU and MirkDC/RSCALE—described in §4.4.

D.1 Sequential MirkDC

Tables 4.7 and 4.8 in §4.4.1 list sixteen experiments designed to measure the relative performance of the four variants of MirkDC on the Sun Ultra 2, and to demonstrate how sequential performance is affected by certain problem and solution strategy parameters. The experiments listed in Table 4.7 are not difficult numerically, in that each SWF-III problem can be solved directly for the specified value of epsilon. The experiments listed in Table 4.8, on the other hand, are more difficult numerically and each requires a form of parameter continuation to achieve convergence. See §4.4.1 for details. This appendix contains numerical results for experiments #5-#8 of each table. Figures summarizing the output of each experiment are indexed in Tables D.1 and D.2. In most cases, the output of an experiment is summarized in two figures: (1) overall and selected program segment execution times, and (2) subroutine call and call-per-mesh profiles. The program segments profiled include five tasks comprising the primary computational costs associated with MirkDC.

The MirkDC solution strategies used in experiments #5-#8 of Table 4.7 are identical to those used in experiments #1-#4 of that table, respectively. Experiments #5-#8 differ only in that the SWF-III problem solved is defined over a wider interval of integration ($[-1, 1]$

Table D.1: Numerical results index for experiments #5-#8 of Table 4.7 in §4.4.1.

exp. #	Results are shown	
	in Figure(s)	on page(s)
5	D.1, D.2	201
6	D.3, D.4	202
7	D.5, D.6	203
8	D.7, D.8	204

Table D.2: Numerical results index for experiments #5-#8 of Table 4.8 in §4.4.1.

exp. #	Results are shown	
	in Figure(s)	on page(s)
5	D.9, D.10, D.17	205, 209
6	D.11, D.12, D.18	206, 209
7	D.13, D.14	207
8	D.15, D.16	208

as opposed to $[0, 1]$). Comparing pairwise Figures (4.27,D.1), (4.28,D.3), (4.30,D.5) and (4.32,D.7), we see that all task execution times are moderately increased. Comparing pairwise Figures (4.26,D.2), (4.29,D.4), (4.31,D.6) and (4.33,D.8), we see that in most cases this increase in execution time can be attributed to a moderate increase in both overall number of subroutine calls and ABD system size. Other effects demonstrated in experiments #5-#8 include the substantial reduction in task execution times (accompanied by an increase in the ratio of ABD matrix construction to factorization time) which can result upon switching to a higher-order MIRK discretization scheme, and the moderate increase in task execution time which can result upon imposing a stricter defect tolerance.

Experiments #5-#8 of Table 4.8 differ from experiments #1-#4 of that table in two respects. First, the SWF-III problem solved is defined over a wider interval of integration ($[-1, 3]$ as opposed to $[0, 2]$). Second, the continuation strategy differs. Specifically, in each of experiments #5-#8, the first two continuation iterations utilize a target value for ϵ approximately 10 times larger than in experiments #1-#4. This change was necessary in order to achieve conver-

gence on the wider interval of integration. Due to the new continuation strategy, however, the wider interval of integration no longer results in increased task execution times as was the case in experiments #5-#8 of Table 4.7. Nevertheless, experiments #5-#8 of Table 4.8 still demonstrate many of the other effects described above. Note that parameter continuation is *necessary* for convergence in these experiments. For example, Figures D.17 and D.18 show the convergence patterns that result when the SWF-III problems in experiments #5 and #6 are solved using MirkDC/COLROW without continuation. MirkDC/COLROW no longer converges in either of these experiments; instead it proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals is exceeded.

Finally, Figure D.15 shows that the ABD matrix construction time of MirkDC/COLROW is greater than that of the other variants in experiment #8. This, of course, is an anomaly. When MirkDC variants exhibit identical convergence patterns in a given experiment—as is always the case in §4.4.1—each of the residual evaluation, defect estimation and ABD matrix construction task execution times should not be noticably different among variants. This is because the algorithm for each of these tasks is identical among variants. Some care was taken to avoid such anomalies by averaging timing results over 4-8 consecutive runs. Occasionally, however, because of unusually high load on a time-shared machine, absolute execution times can be consistently inflated over 4-8 consecutive runs. This certainly is the case in experiment #8 of Table 4.8, and also in any other experiment presented in §4.4.1 where there is a noticable difference among variants in the residual evaluation, defect estimation or ABD matrix construction task execution times.

Figure D.1: Overall and selected program segment execution times of the four variants of MirkDC in experiment #5 of Table 4.7. The slightly wider disk spacing results in a small increase in execution times over experiment #1. Compare to Figure 4.27, and compare the profiles in Figures 4.26 and D.2.

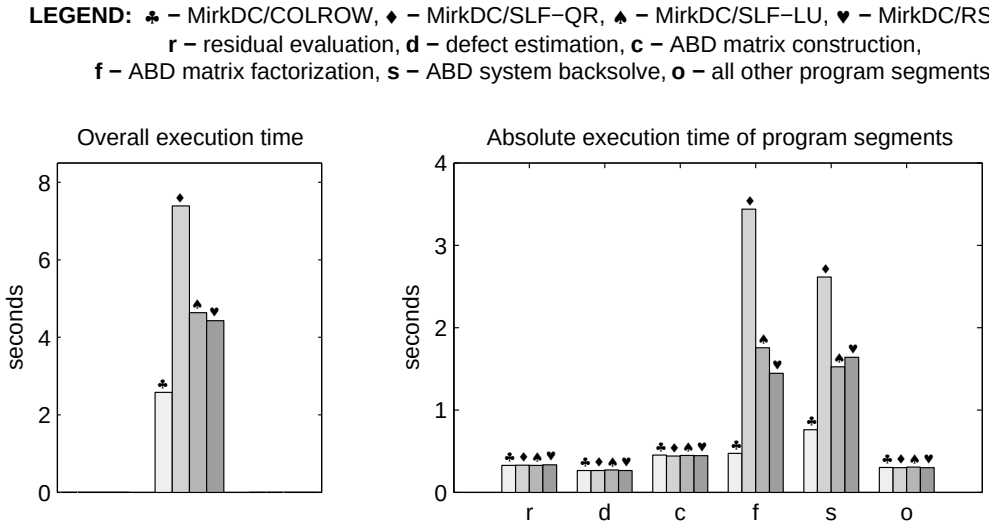


Figure D.2: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #5 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

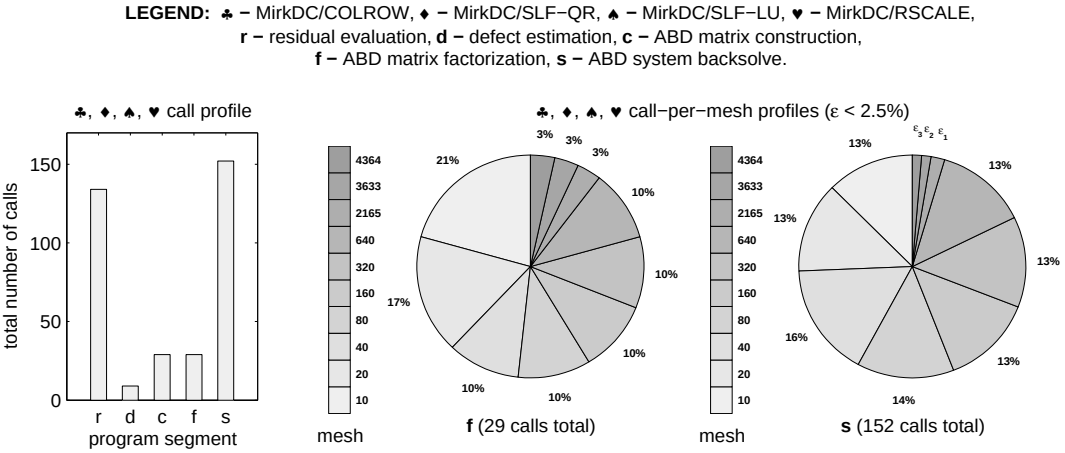


Figure D.3: Overall and selected program segment execution times of the four variants of MirkDC in experiment #6 of Table 4.7. Switching to a 4^{th} order MIRK scheme results in a substantial reduction in execution times. Compare to Figure D.1, and compare the profiles in Figures D.2 and D.4.

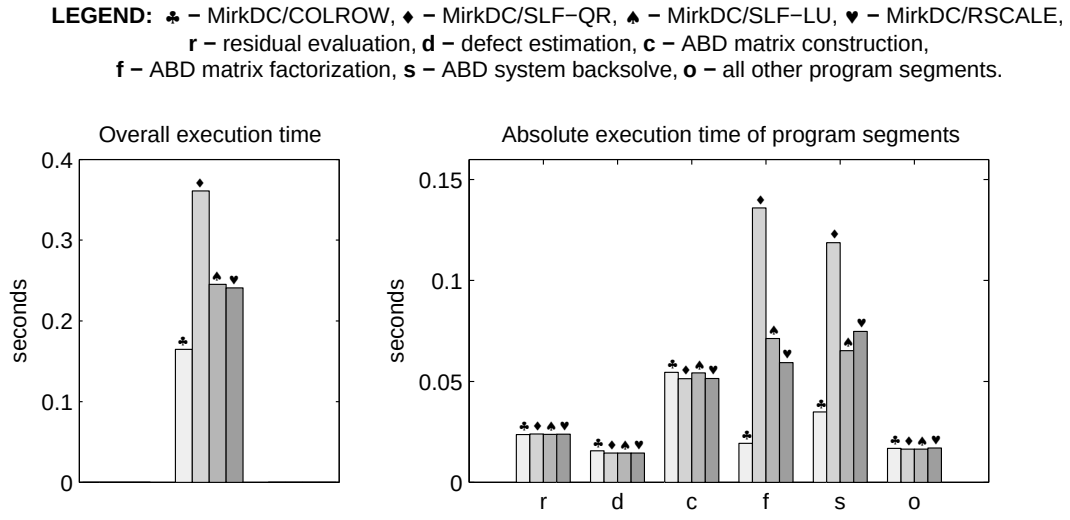


Figure D.4: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #6 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

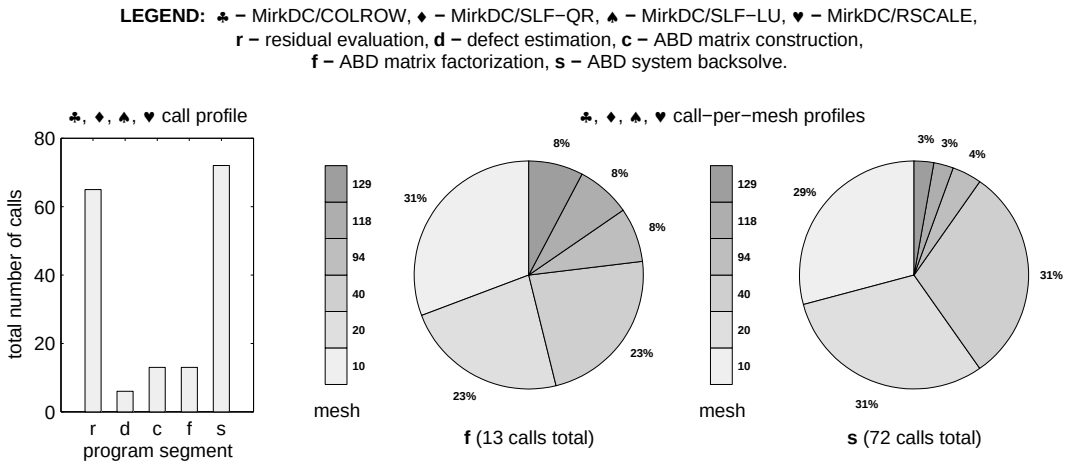


Figure D.5: Overall and selected program segment execution times of the four variants of MirkDC in experiment #7 of Table 4.7. Switching to a stricter defect tolerance results in a moderate increase in execution times. Compare to Figure D.3, and compare the profiles in Figures D.4 and D.6.

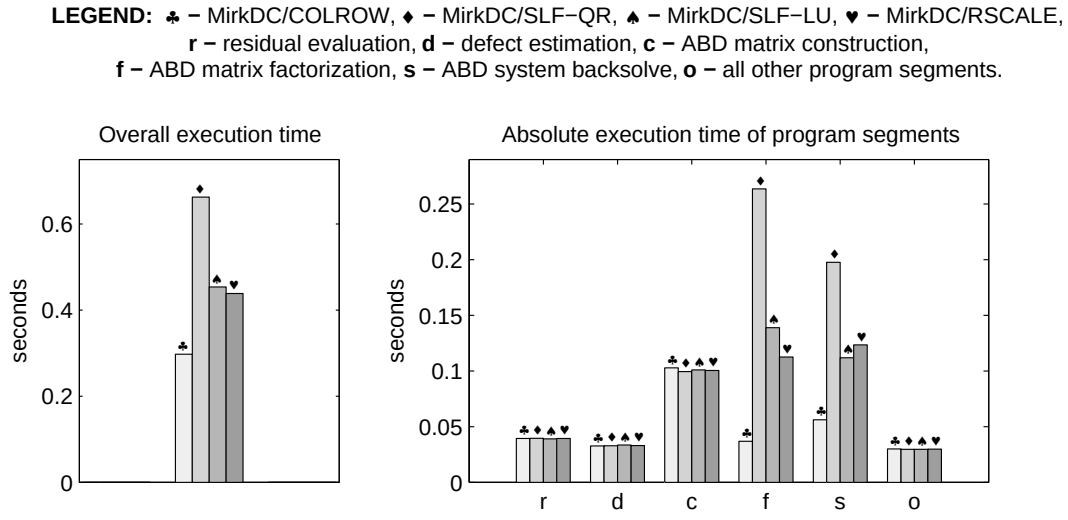


Figure D.6: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #7 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

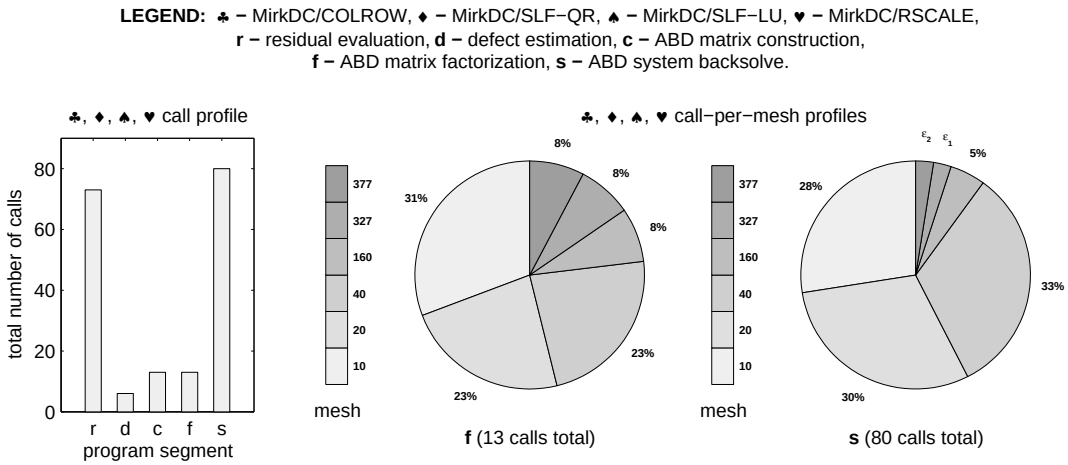


Figure D.7: Overall and selected program segment execution times of the four variants of MirkDC in experiment #8 of Table 4.7. Switching to a 6^{th} order MIRK scheme results in a moderate reduction in execution times. Compare to Figure D.5, and compare the profiles in Figures D.6 and D.8.

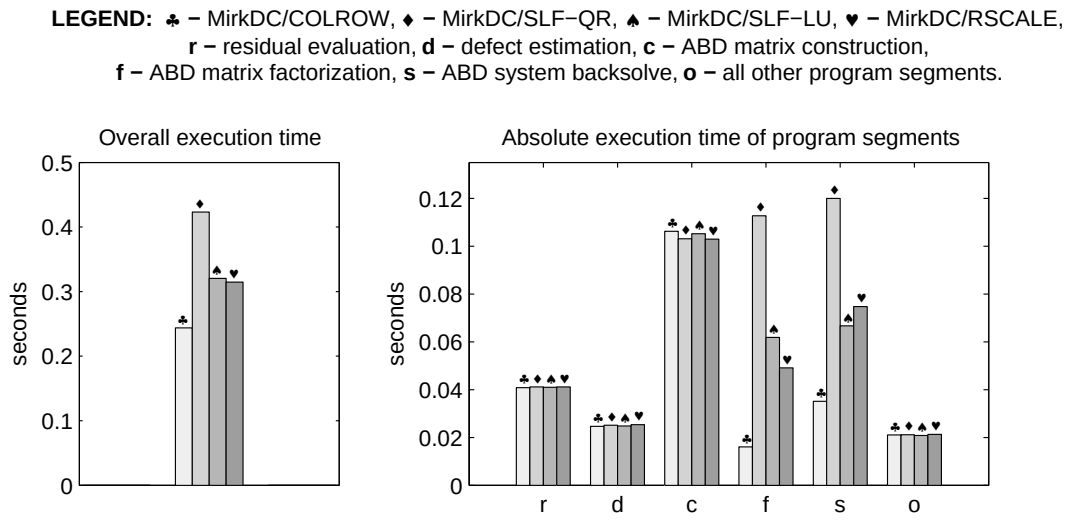


Figure D.8: Subroutine call and call-per-mesh profiles of the four variants of MirkDC in experiment #8 of Table 4.7. Each variant succeeds in satisfying the specified τ_{defect}. Profiles are identical among variants.

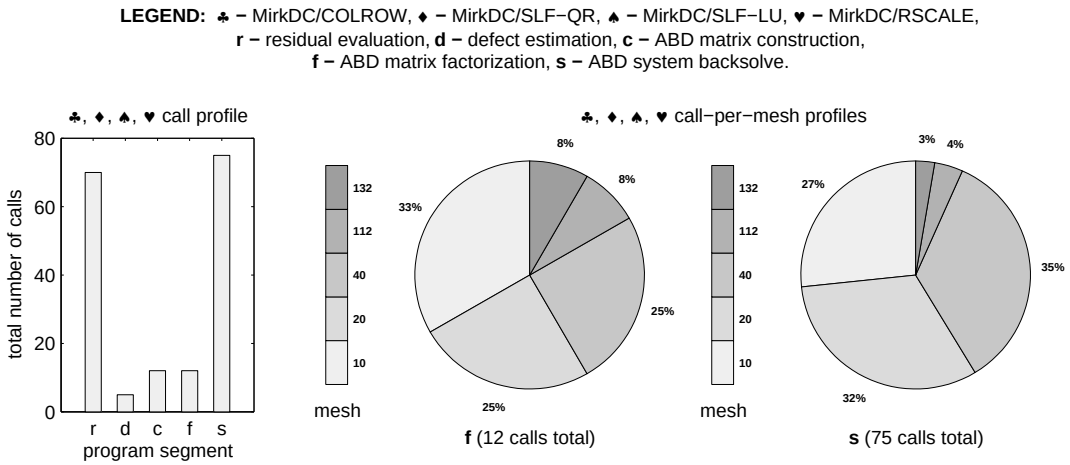


Figure D.9: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #5 of Table 4.8. Although the disk spacing is wider than in experiment #1, the new continuation strategy employed in experiment #5 actually results in a moderate decrease in execution times. Compare to Figure 4.36, and compare the profiles in Figures 4.35 and D.10.

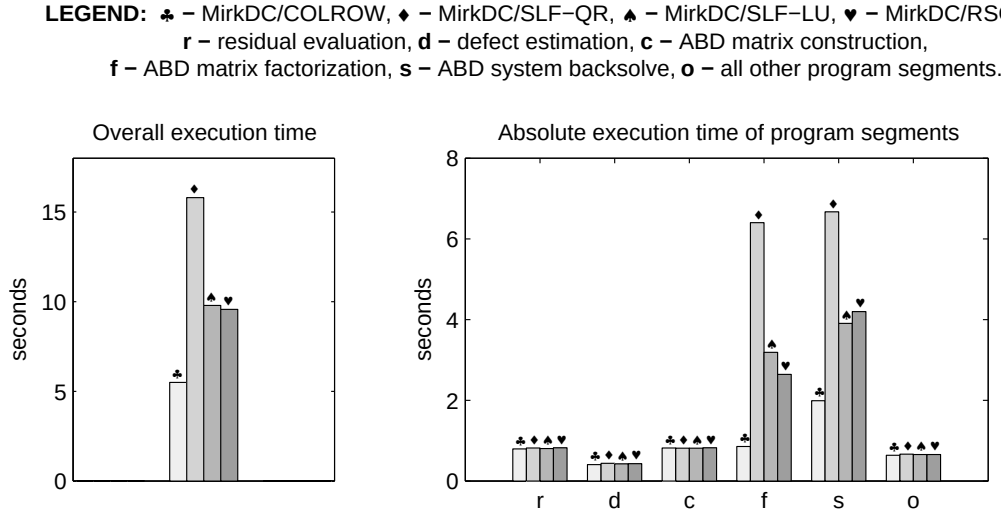


Figure D.10: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #5 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 988, 1464 and 4752 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

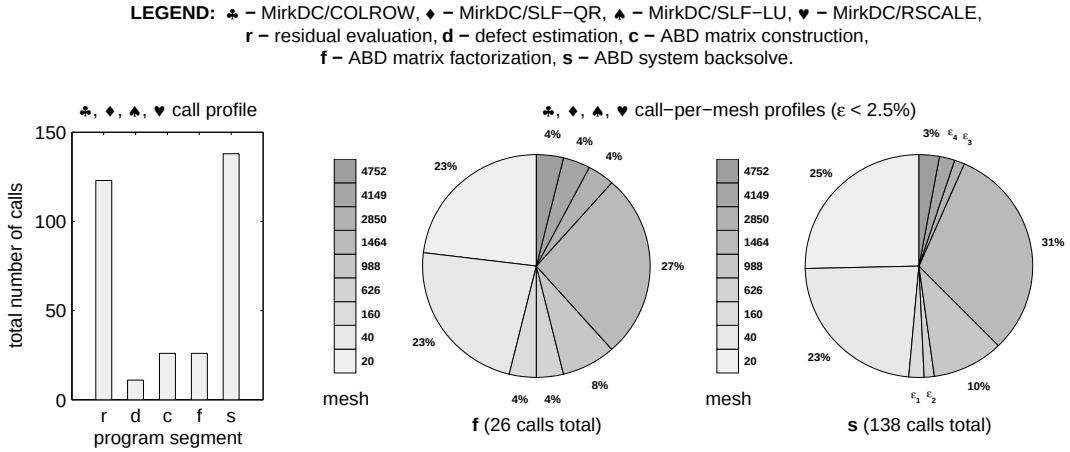


Figure D.11: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #6 of Table 4.8. Switching to a 4^{th} order MIRK scheme results in a substantial reduction in execution times, even with the stricter defect tolerance. Compare to Figure D.9, and compare the profiles in Figures D.10 and D.12.

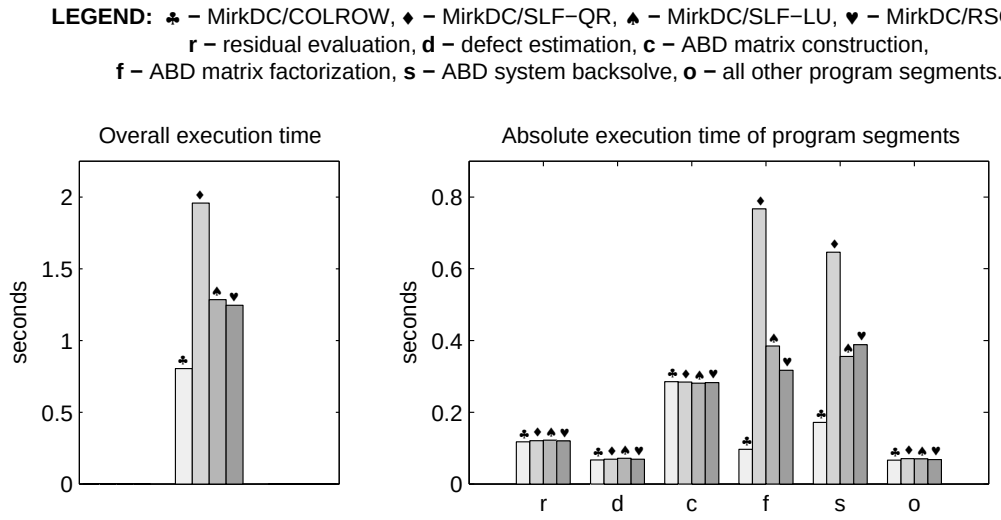


Figure D.12: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #6 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 119, 169 and 478 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

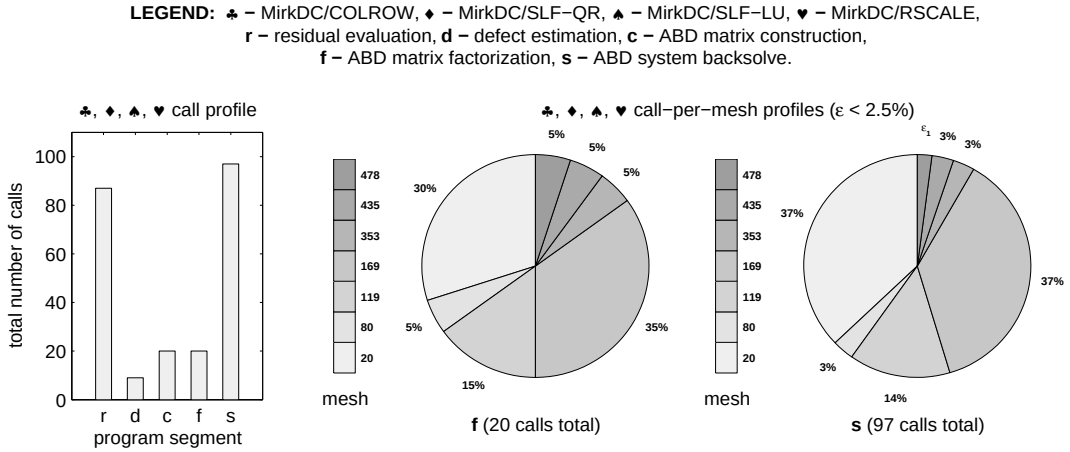


Figure D.13: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #7 of Table 4.8. Switching to a stricter defect tolerance results in a moderate increase in execution times. Compare to Figure D.11, and compare the profiles in Figures D.12 and D.14.

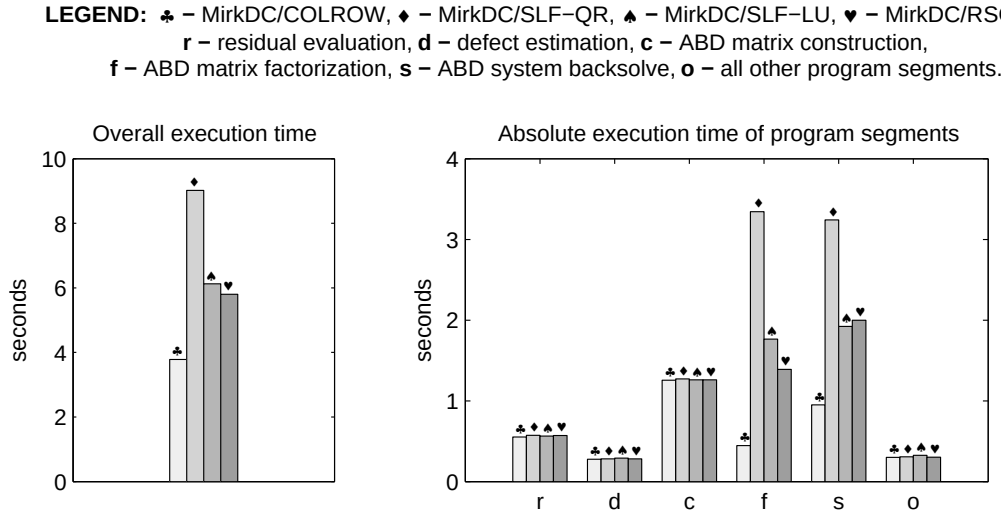


Figure D.14: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #7 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 625, 933 and 2434 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

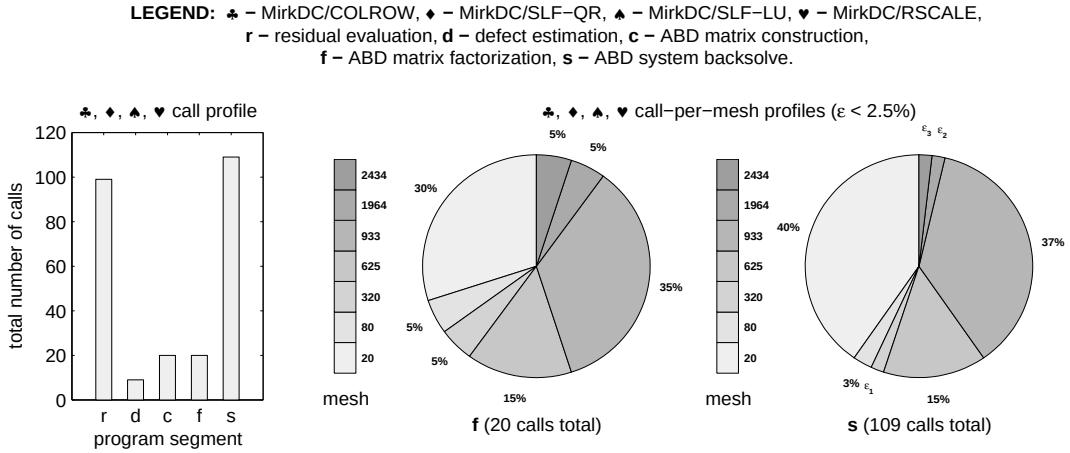


Figure D.15: Overall and selected program segment execution times, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #8 of Table 4.8. Switching to a 6^{th} order MIRK scheme results in a moderate reduction in execution times. Compare to Figure D.13, and compare the profiles in Figures D.14 and D.16. (Note: The seemingly higher ABD matrix construction time of MirkDC/COLROW is just a timing anomaly; see §D.1 for details.)

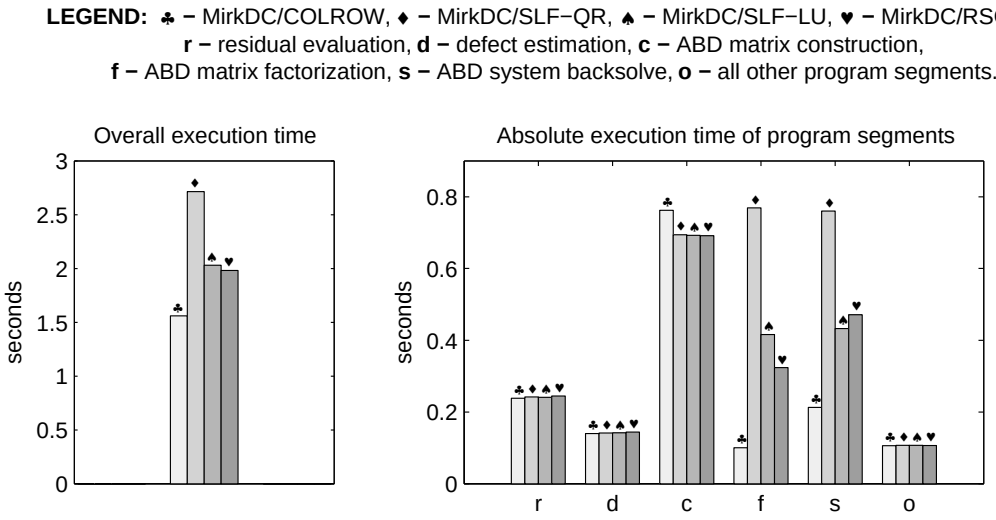


Figure D.16: Subroutine call and call-per-mesh profiles, accumulated over all three continuation iterations, of the four variants of MirkDC in experiment #8 of Table 4.8. The first, second and third continuation iteration terminates with a final mesh of 151, 215 and 509 subintervals, respectively. Each variant succeeds in satisfying the specified τ_{defect} at each iteration. Profiles are identical among variants.

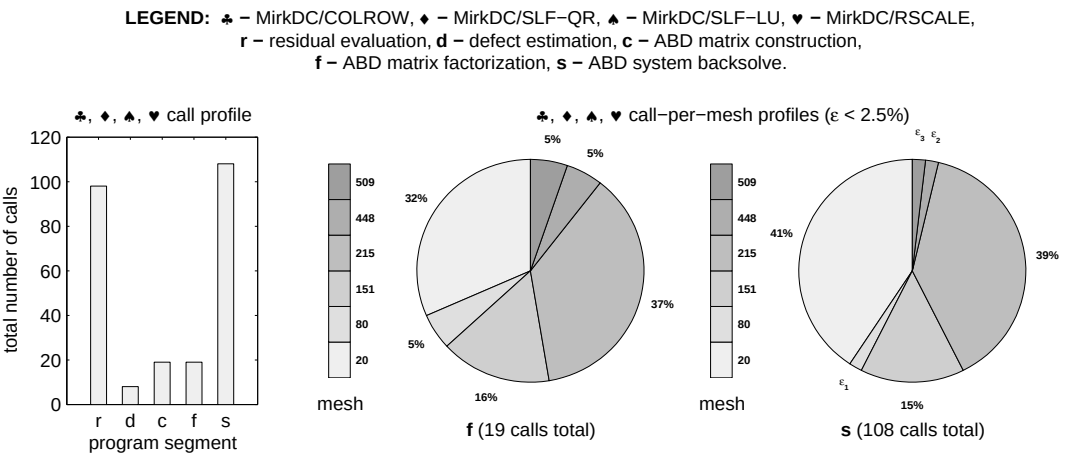


Figure D.17: Subroutine call and call-per-mesh profiles of MirkDC/COLROW in experiment #5 of Table 4.8, *when parameter continuation is not used*. Without continuation, MirkDC/COLROW does not converge. The code proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals is exceeded; the defect estimation stage is never reached.

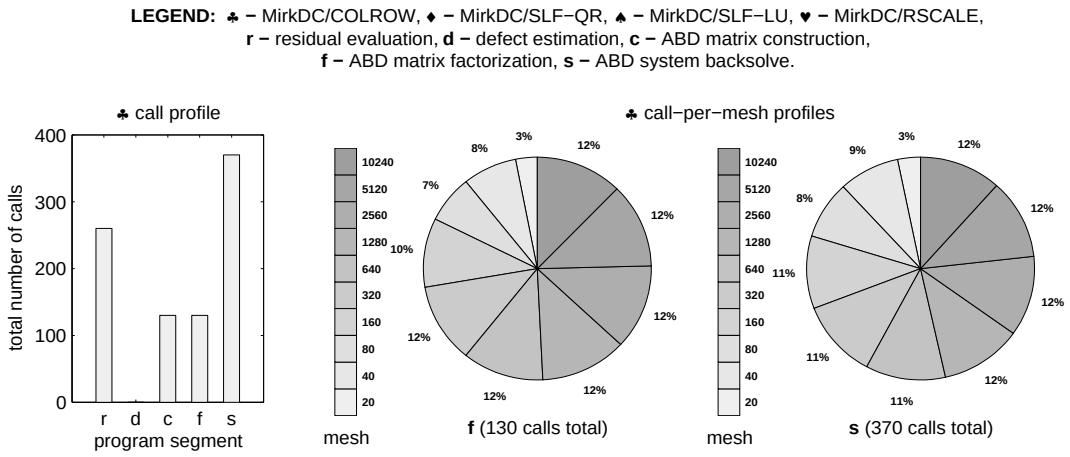


Figure D.18: Subroutine call and call-per-mesh profiles of MirkDC/COLROW in experiment #6 of Table 4.8, *when parameter continuation is not used*. Without continuation, MirkDC/COLROW does not converge. The code proceeds through a sequence of failed Newton iterations followed by mesh doubling until the maximum number of subintervals is exceeded; the defect estimation stage is never reached.

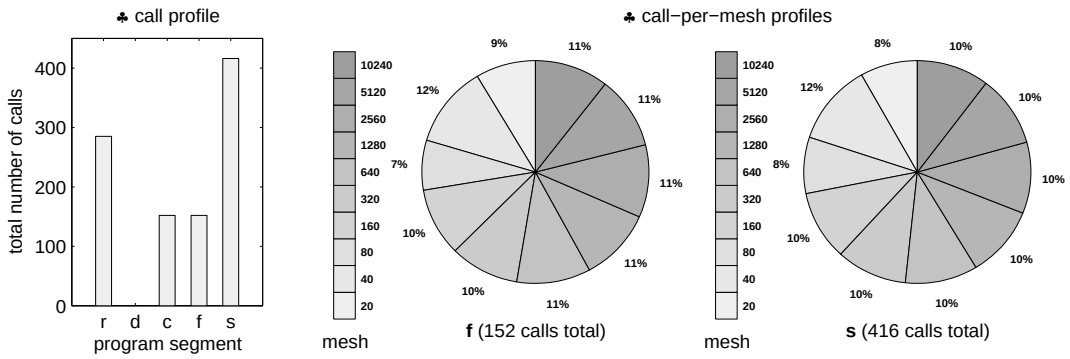


Table D.3: Numerical results index for experiments in Table 4.10 in §4.4.2.

exp. #	Results are shown	
	in Figure(s)	on page(s)
1	D.19, D.20	211
3	D.21, D.22	212
5	D.23, D.24, D.25	213, 214
6	D.26, D.27, D.28	215, 216
7	D.29, D.30, D.31	217, 218
9	D.32, D.34, D.35	219, 220
10	D.33, D.36, D.37	219, 221

D.2 Parallel MirkDC

Table 4.10 in §4.4.2 lists ten experiments demonstrating the parallel (and sequential) performance of PMirkDC, the parallel implementation of MirkDC/RSCALE appearing in [Muir 03]. This appendix contains numerical results for experiments #1, #3, #5, #6, #7, #9 and #10. Figures summarizing the output of each experiment are indexed in Table D.3.

In experiments #1 and #3, the execution time required to solve Problem A of Table 4.9 is substantially increased by choosing an uneccessarily fine initial mesh of $M_0 = 7000$ subintervals. While execution times exhibit better speedups, these experiments are not realistic as MirkDC does not require such a fine initial mesh to achieve convergence. In experiments #5, #6 and #7, more realistic computationally intensive experiments are performed by solving more difficult SWF-III problems with the help of parameter continuation.

In experiments #9 and #10, we attempt to solve difficult SWF-III problems without using parameter continuation. In both experiments, the convergence pattern of PMirkDC is shorter than that of MirkDC, resulting in a substantial improvment in execution time even on a sequential machine.

Figure D.19: Overall speedup and execution time of MirkDC and PMirkDC in experiment #1 of Table 4.10. Parallelism begins to pay-off at 2 processors.

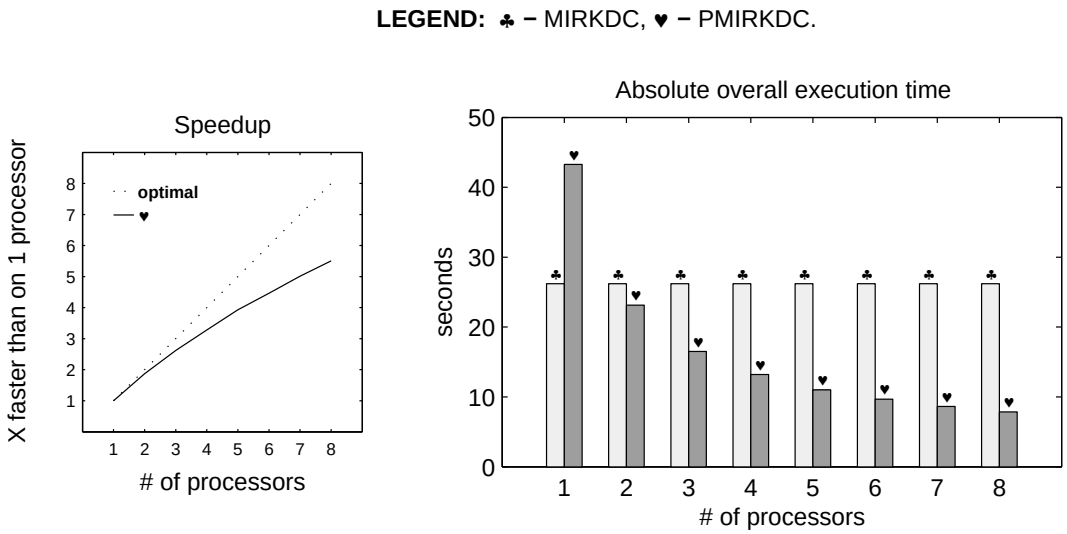


Figure D.20: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #1 of Table 4.10.

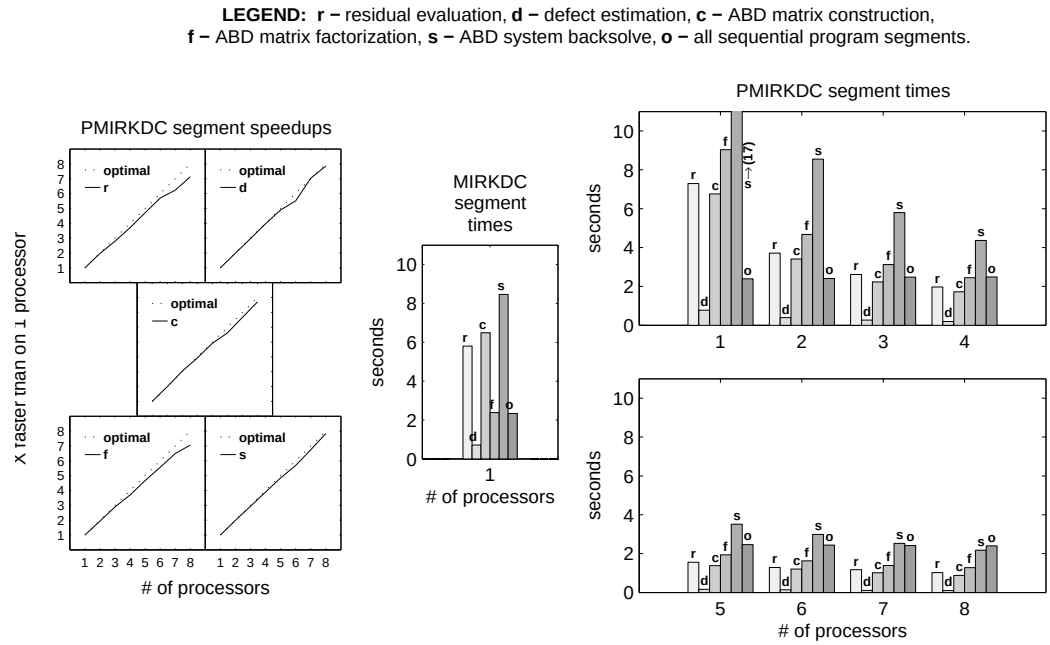


Figure D.21: Overall speedup and execution time of MirkDC and PMirkDC in experiment #3 of Table 4.10. The same problem is solved as in experiment #1, using the same solution strategy, but this time on the Origin 2000 instead of the Challenge L. Execution times are nearly 5 times faster (compare to Figure D.19). Parallelism begins to pay-off at 3 processors.

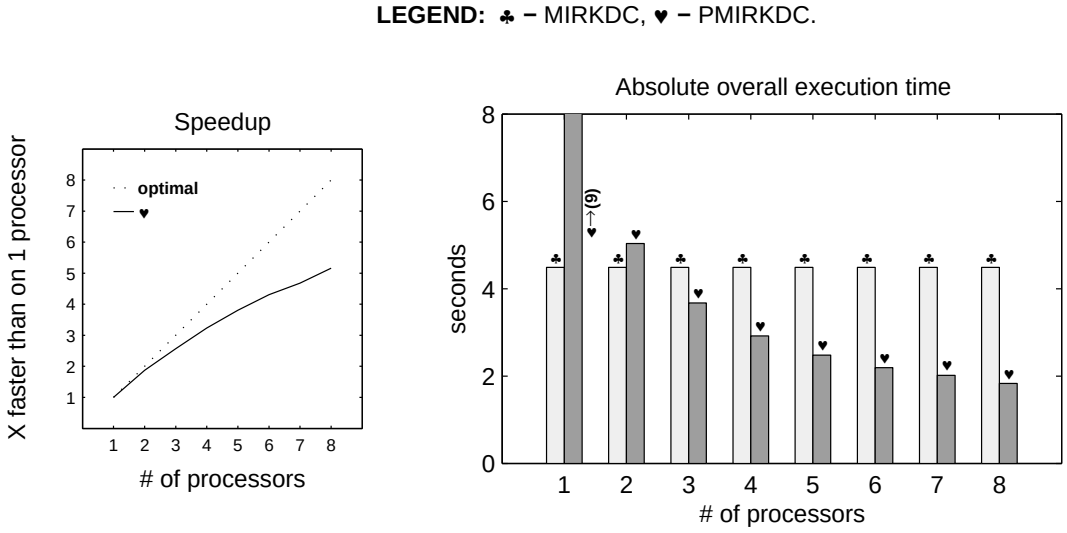


Figure D.22: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #3 of Table 4.10. Compare to Challenge L results shown in Figure D.20.

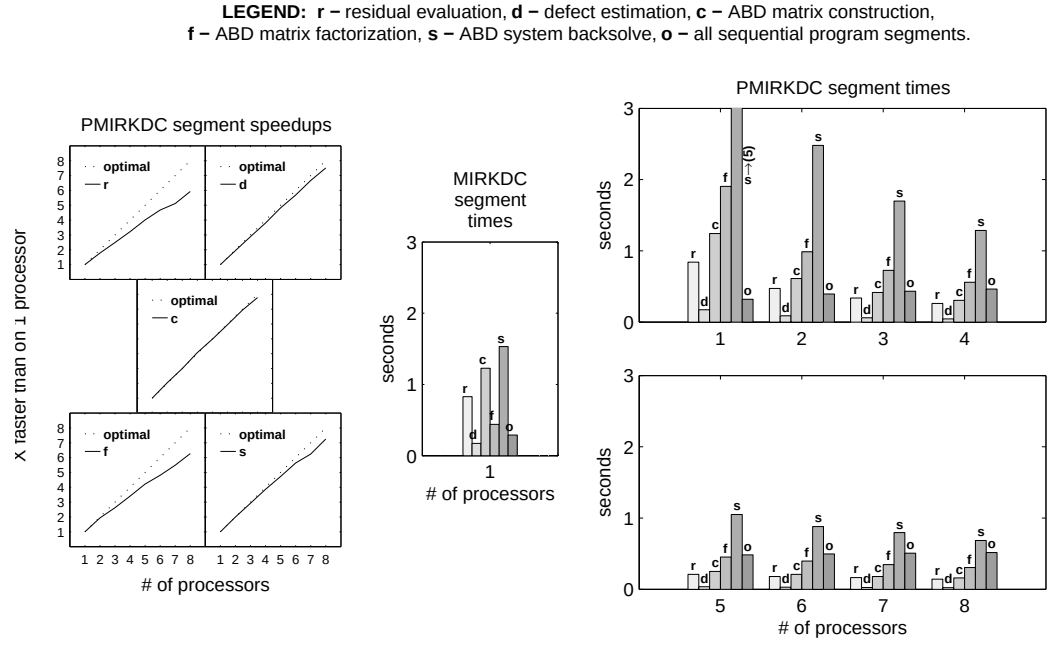


Figure D.23: Overall speedup and execution time of MirkDC and PMirkDC in experiment #5 of Table 4.10. Results are shown for the final continuation step only. Parallelism begins to pay-off at 3 processors.

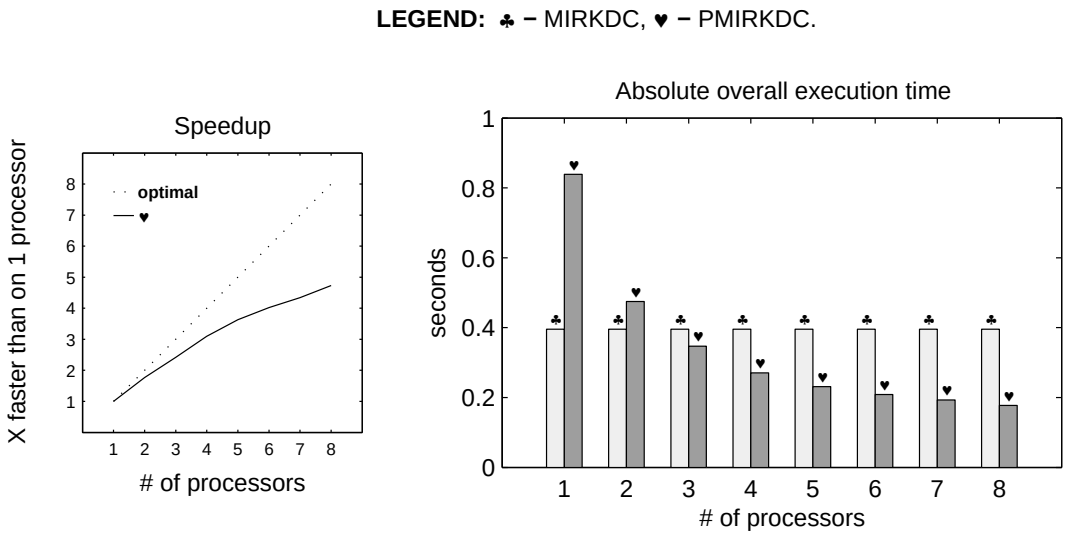


Figure D.24: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #5 of Table 4.10. Results are shown for the final continuation step only.

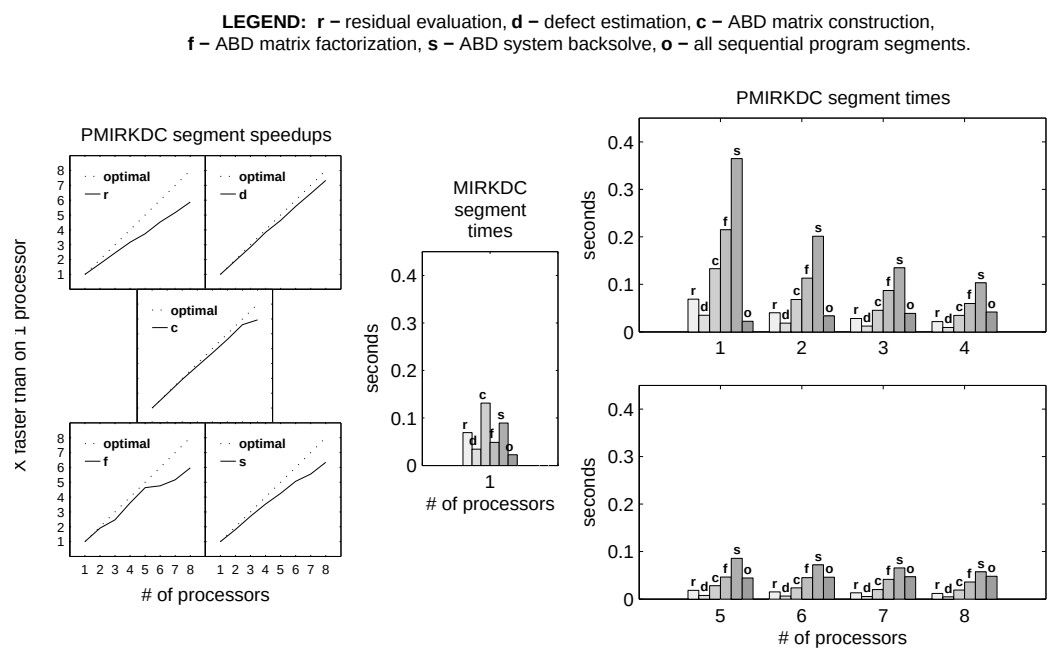


Figure D.25: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #5 of Table 4.10. Results are accumulated over all five continuation steps.

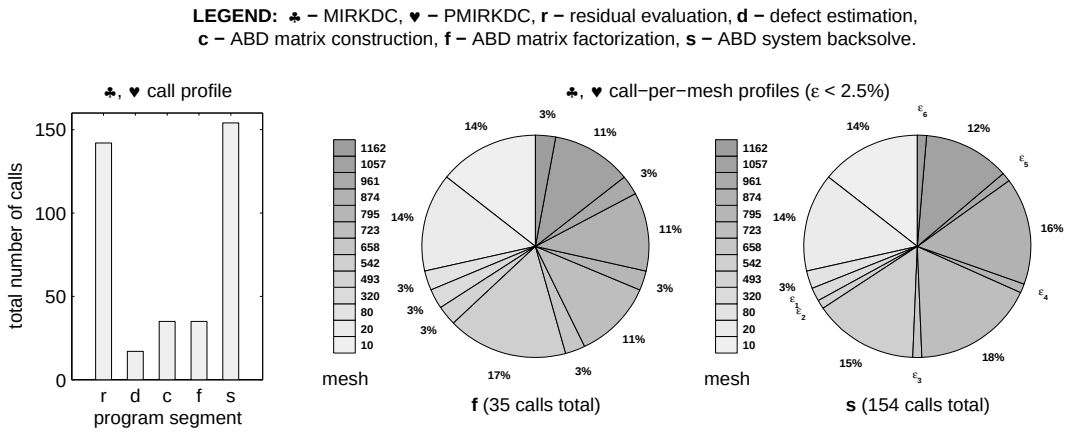


Figure D.26: Overall speedup and execution time of MirkDC and PMirkDC in experiment #6 of Table 4.10. Results are shown for the final continuation step only. Parallelism begins to pay-off at 3 processors.

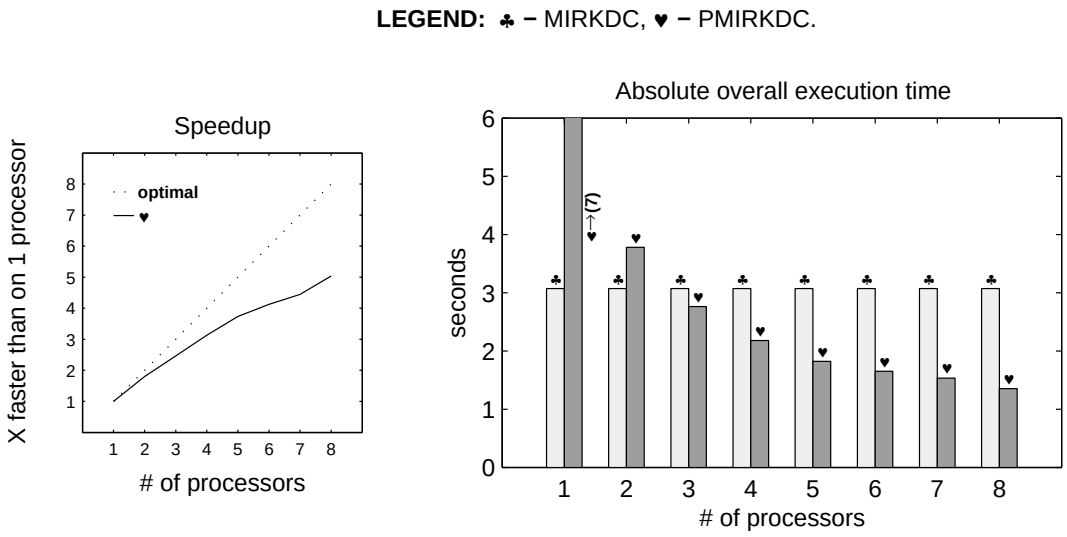


Figure D.27: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #6 of Table 4.10. Results are shown for the final continuation step only.

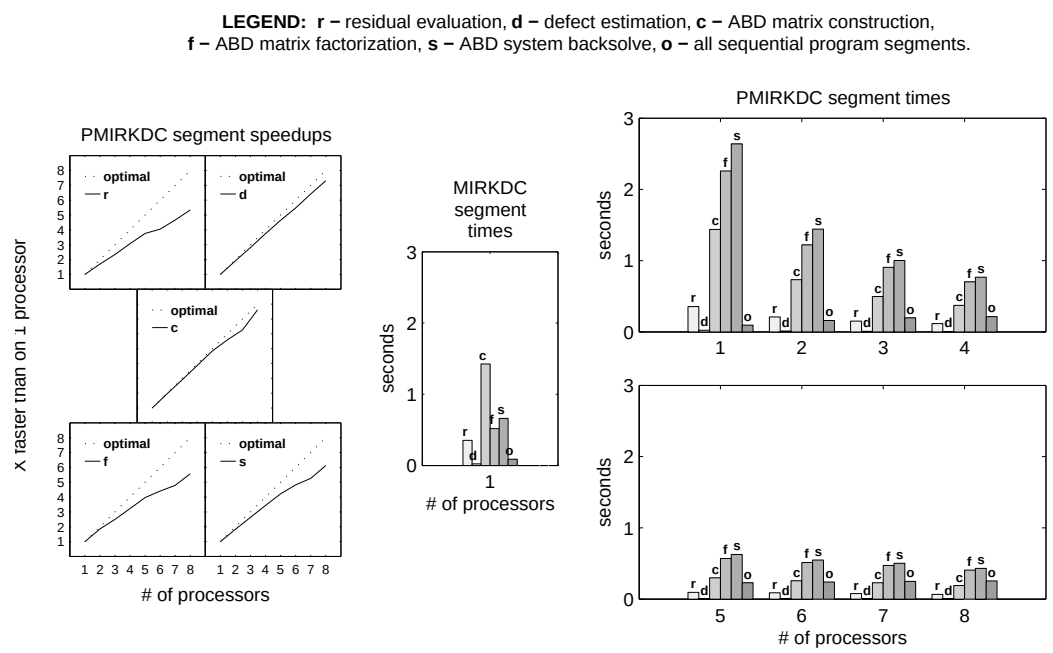


Figure D.28: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #6 of Table 4.10. Results are accumulated over all five continuation steps.

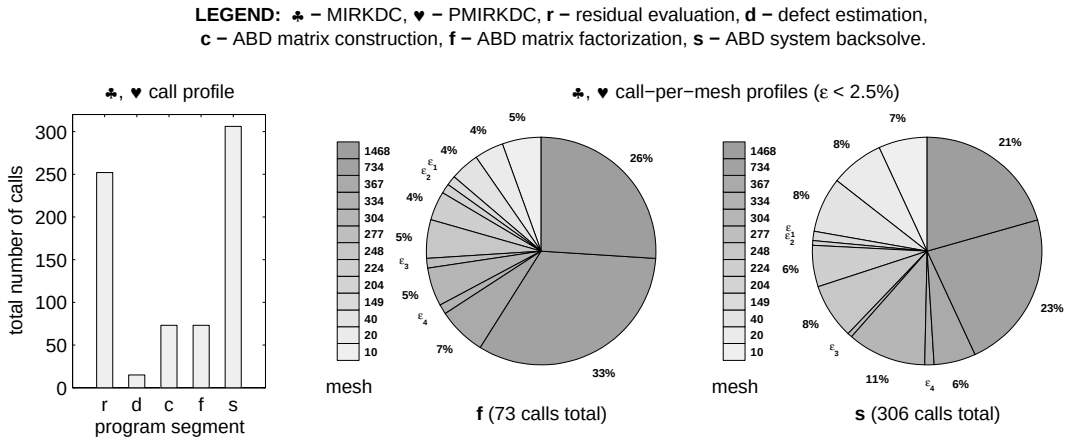


Figure D.29: Overall speedup and execution time of MirkDC and PMirkDC in experiment #7 of Table 4.10. Results are shown for the final continuation step only. Parallelism begins to pay-off at 3 processors.

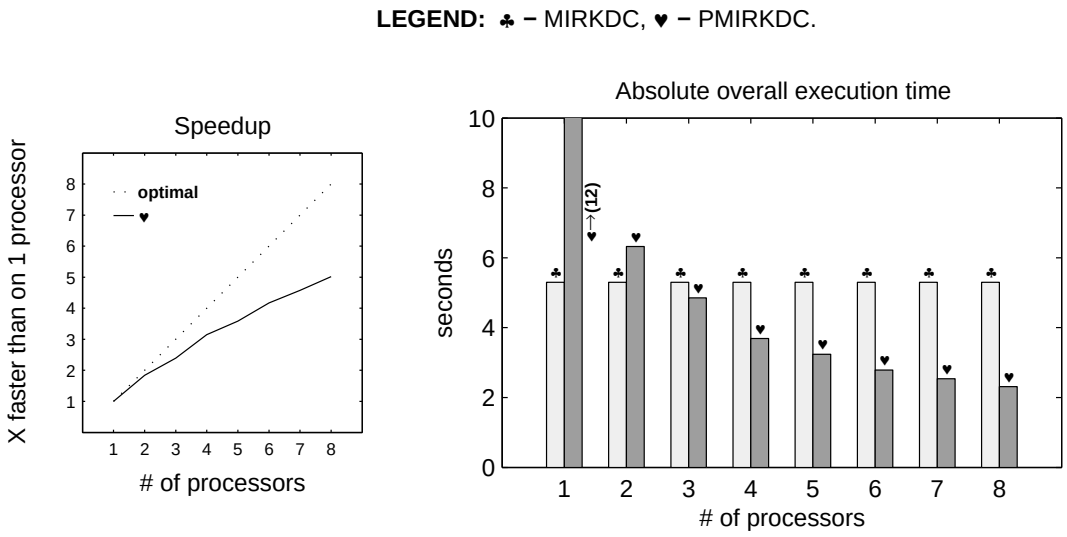


Figure D.30: Speedup and execution time of selected program segments of MirkDC and PMirkDC in experiment #7 of Table 4.10. Results are shown for the final continuation step only.

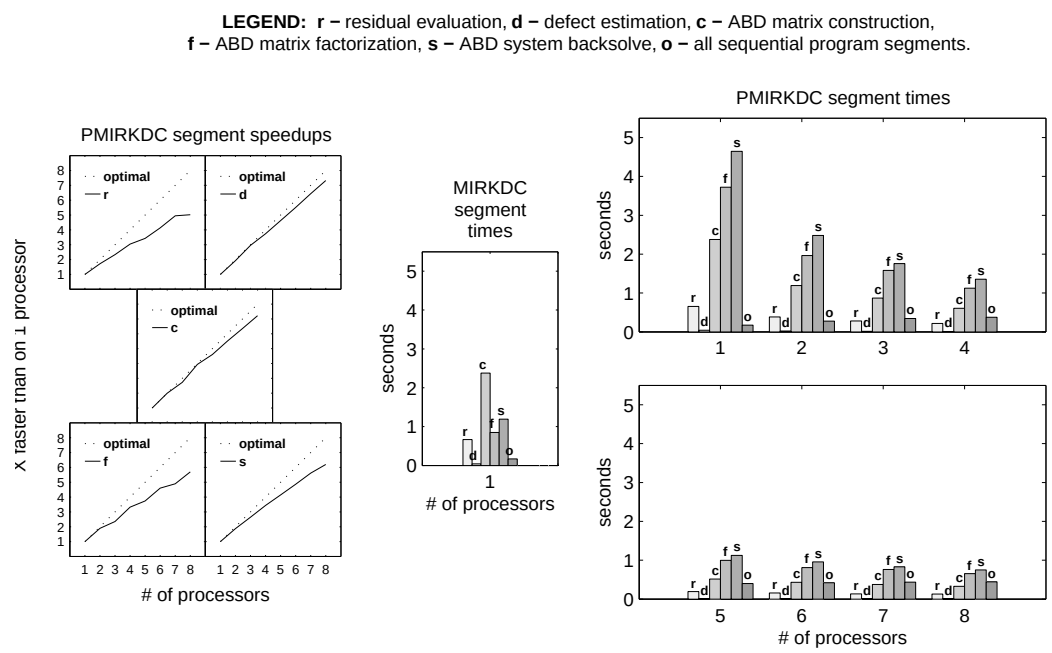


Figure D.31: Subroutine call and call-per-mesh profiles of MirkDC and PMirkDC in experiment #7 of Table 4.10. Results are accumulated over all five continuation steps.

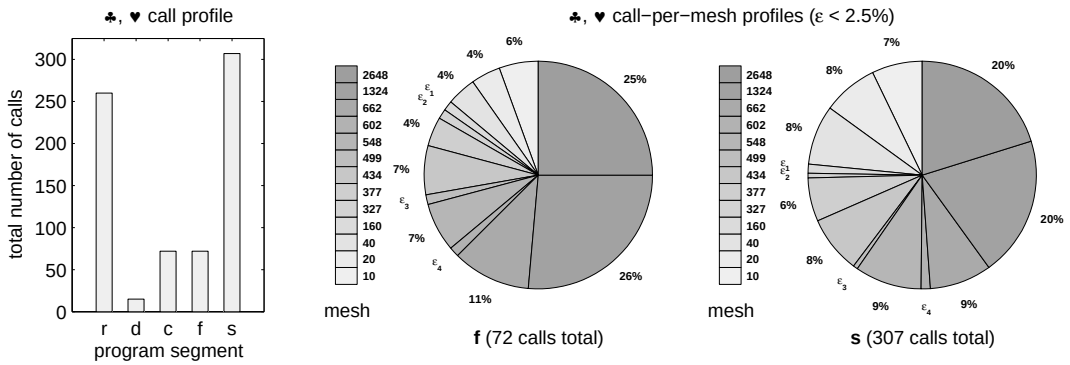


Figure D.32: Overall and selected program segment execution times of MirkDC and PMirkDC in experiment #9 of Table 4.10. This experiment is run sequentially on the Ultra 2. The vast difference in execution times is explained by the subroutine call and call-per-mesh profiles shown in Figures D.34 and D.35.

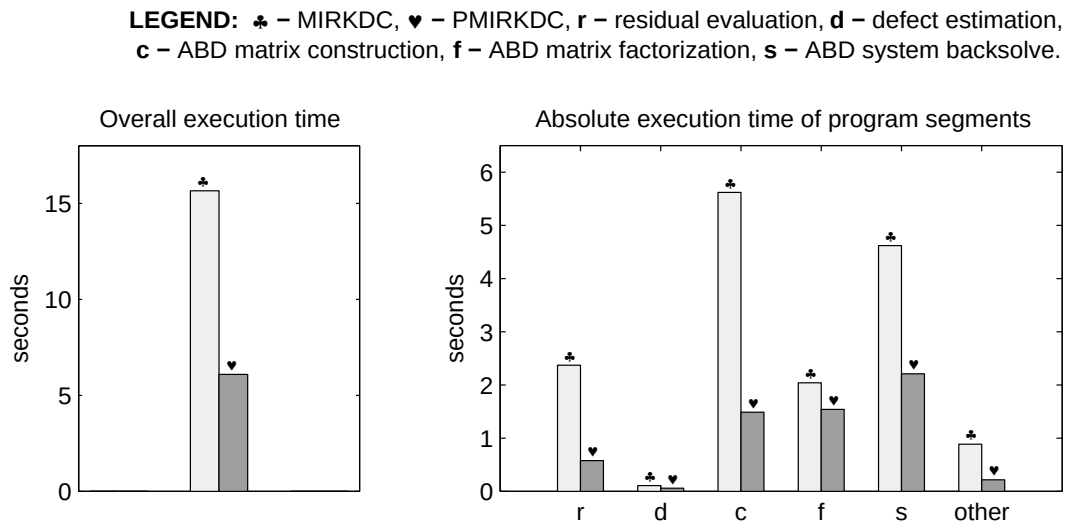


Figure D.33: Overall and selected program segment execution times of MirkDC and PMirkDC in experiment #10 of Table 4.10. This experiment is run sequentially on the Ultra 2. The vast difference in execution times is explained by the subroutine call and call-per-mesh profiles shown in Figures D.36 and D.37.

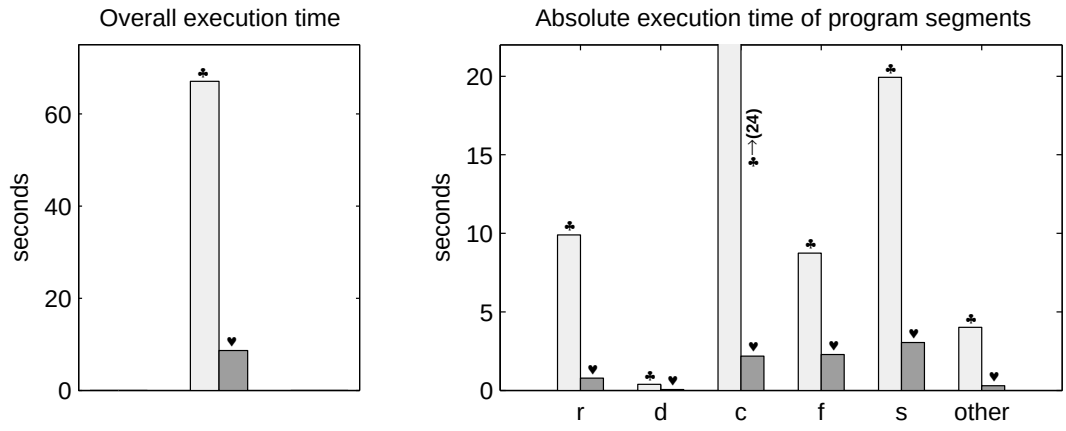


Figure D.34: Profiles of MirkDC in experiment #9 of Table 4.10. These differ significantly from those of PMirkDC (Figure D.35).

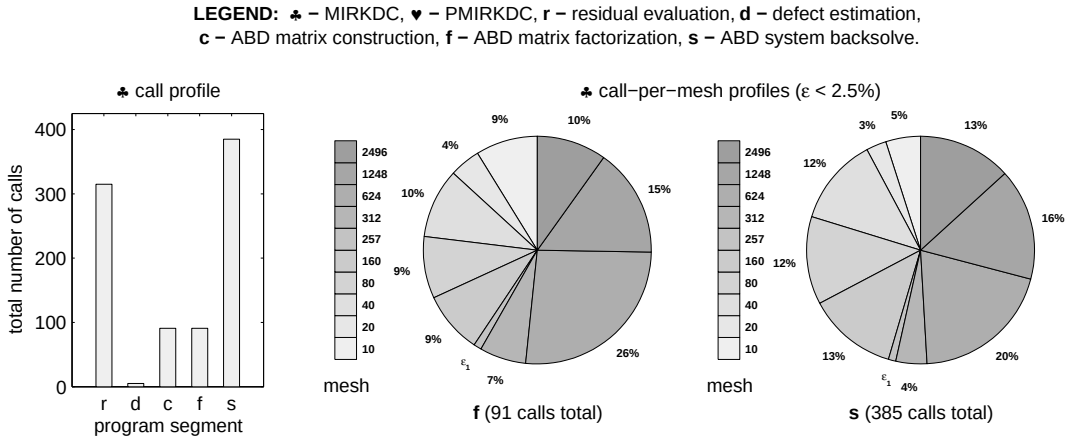


Figure D.35: Profiles of PMirkDC in experiment #9 of Table 4.10. PMirkDC computes an acceptable solution using fewer subroutine calls and fewer mesh subintervals than MirkDC resulting in substantially reduced execution time (Figure D.32).

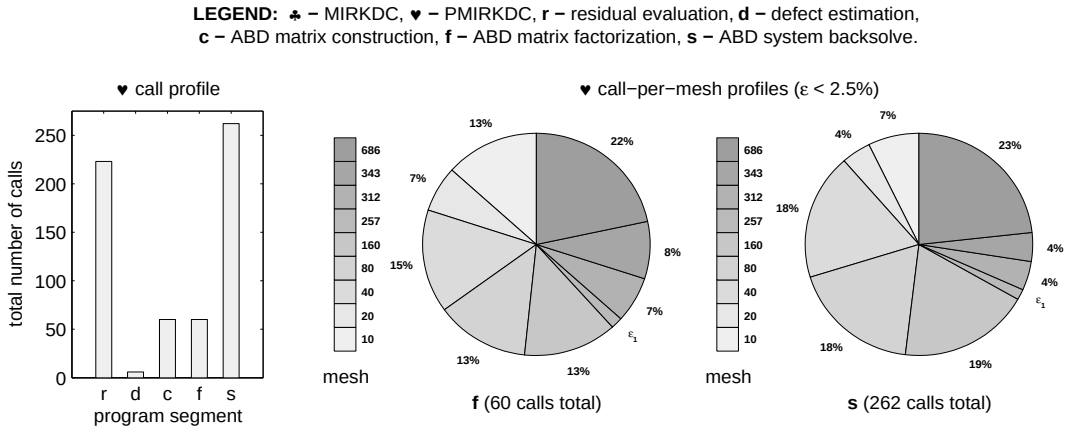


Figure D.36: Profiles of MirkDC in experiment #10 of Table 4.10. These differ significantly from those of PMirkDC (Figure D.37).

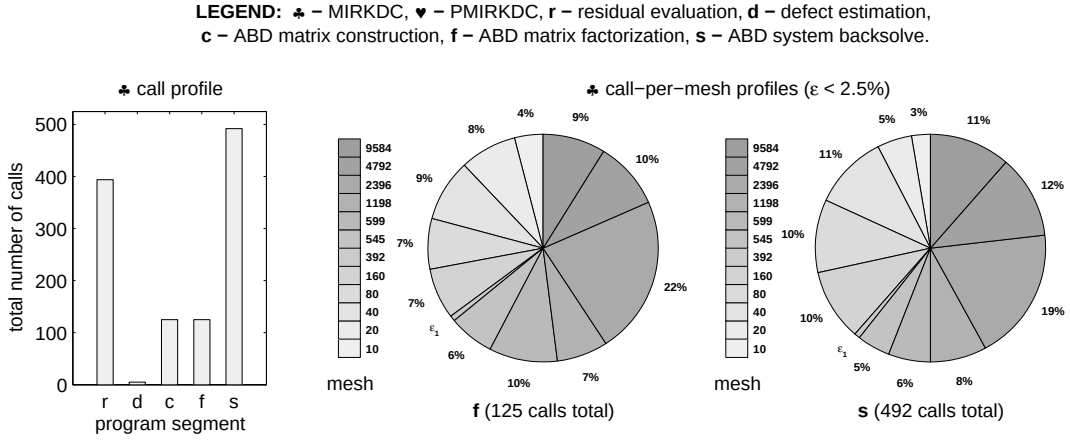


Figure D.37: Profiles of PMirkDC in experiment #10 of Table 4.10. PMirkDC computes an acceptable solution using fewer subroutine calls and fewer mesh subintervals than MirkDC resulting in substantially reduced execution time (Figure D.33).

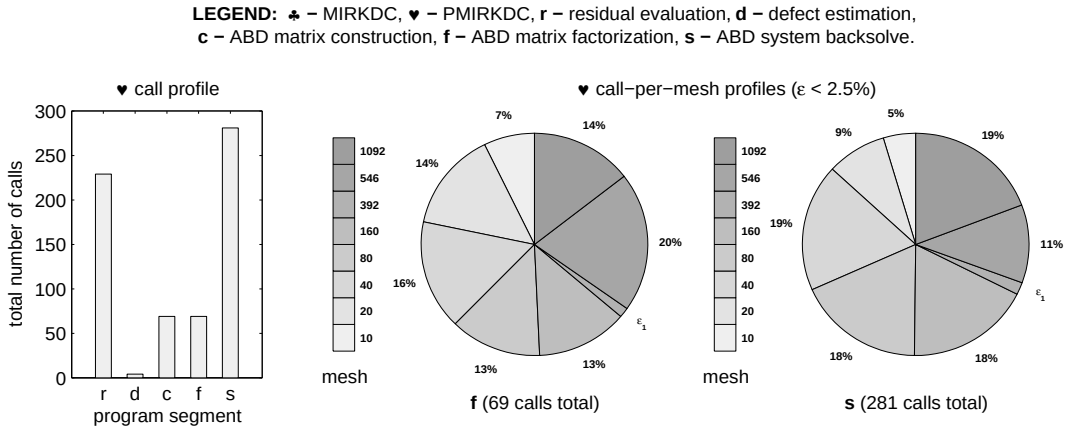


Table D.4: Numerical results index for experiments #3-#8 of Table 4.11 in §4.4.3.

exp. #	Results are shown	
	in Figure(s)	on page(s)
3	D.38, D.39, D.40, D.41	223, 224
4	D.42, D.39, D.43, D.44	223, 225, 226
5	D.45, D.39, D.46, D.47, D.57, D.58	223, 227, 228, 234
6	D.48, D.49	229
7	D.50, D.51, D.52, D.53, D.59	230, 231, 235
8	D.54, D.51, D.55, D.56, D.60	230, 232, 233, 235

D.3 Problems Where MirkDC/*SLF*-LU Fails

Table 4.11 in §4.4.3 lists eight experiments designed to show how *SLF*-LU instability can affect MirkDC performance; this appendix contains numerical results for experiments #3-#8 of that table. Figures summarizing the output of each experiment are indexed in Table D.4. In most cases, the output of an experiment is summarized in four figures: (1) overall and selected program segment execution times, (2) profiles of the stable variants of MirkDC, (3) profiles of MirkDC/*SLF*-LU, and (4) execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LUsolution.

Of particular interest in these results are Figures D.41, D.44, D.47 and D.53, where we compare the execution time and accuracy of the three solvers. In each of these figures, there are one or more examples where *SLF*-LU exhibits instability when used to solve a well-conditioned ABD system—a system for which the other solvers have no difficulty finding a solution. Figure D.56 shows some results with poorly-conditioned ABD systems. Figures D.57-D.60 show the effects of partitioning on solver accuracy. In each of these experiments, we solve and resolve a selected system using a varying number of partitions. Only the stability of *SLF*-LU seems affected by the partitioning strategy. Surprisingly, single-partitioned *SLF*-LU is stable on these problems.

Figure D.38: Overall and selected program segment execution times of the four variants of MirkDC in experiment #3 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures D.39 and D.40.

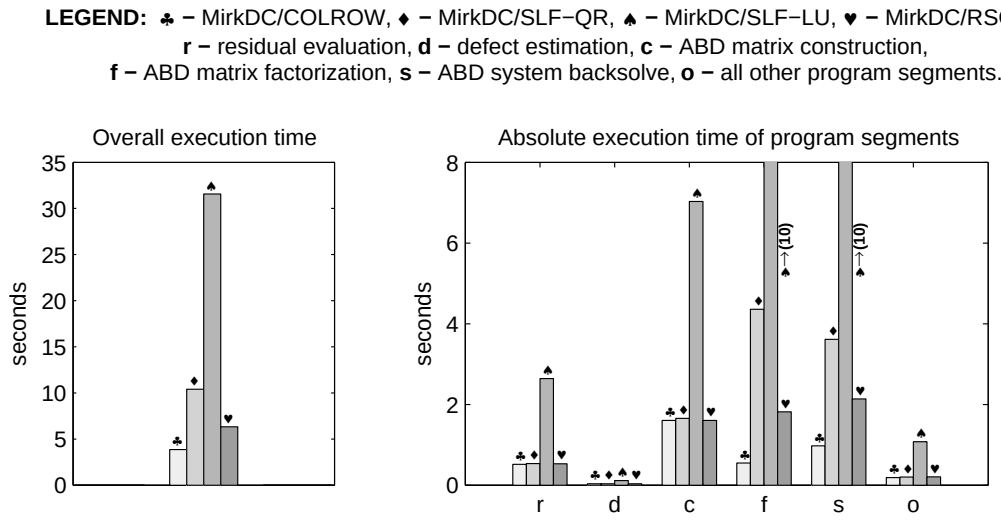


Figure D.39: Profiles of the *stable* variants of MirkDC (MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/RSCALE) in experiments #3, #4, and #5 of Table 4.11. Each of these variants converges in all three experiments. Profiles are identical among variants and experiments.

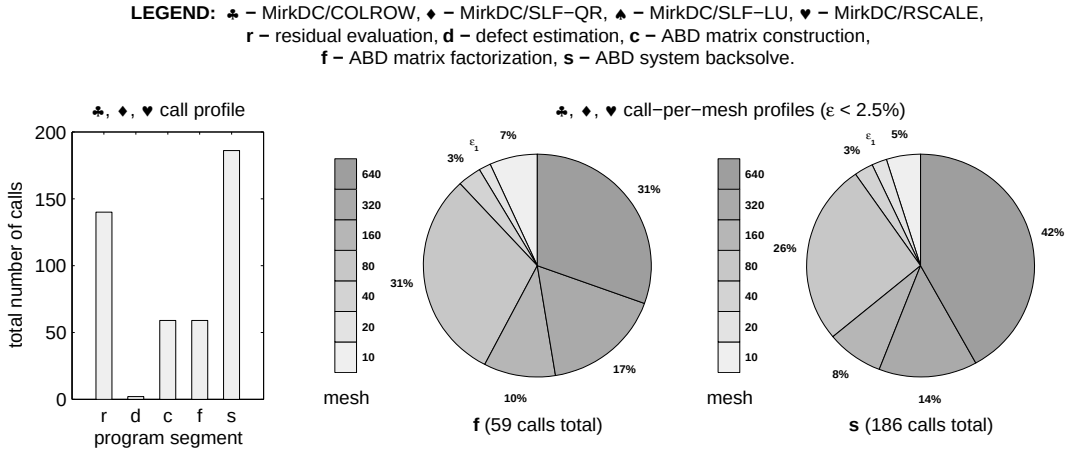


Figure D.40: Profiles of MirkDC/*SLF*-LU in experiment #3 of Table 4.11. This variant of MirkDC converges in this experiment, albeit less efficiently than the other three variants (Figure D.39). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 640 and 1280. This instability is investigated further in Figure D.41.

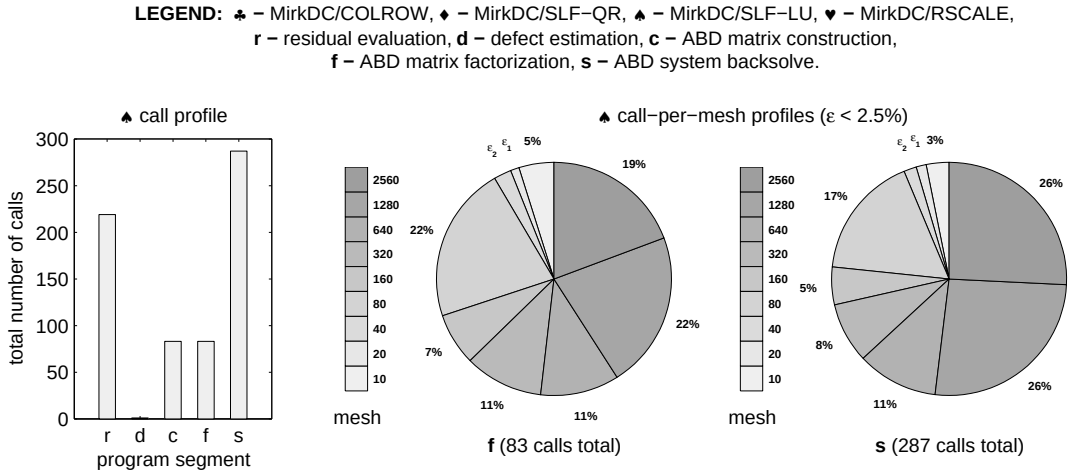


Figure D.41: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #3 (Figure D.40). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 9@mesh 640 and 7@mesh 1280 (selections #3 and #6).

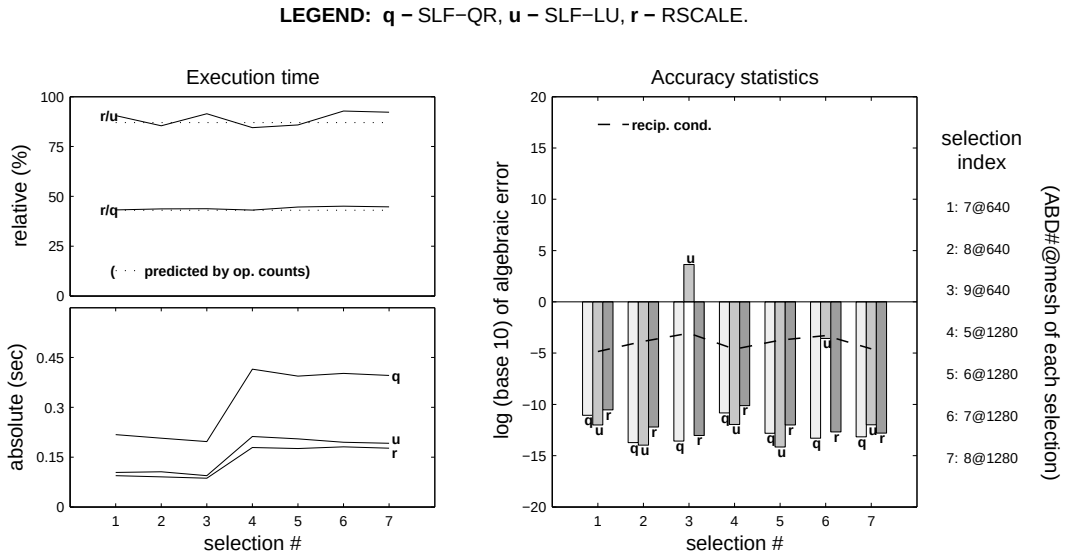


Figure D.42: Overall and selected program segment execution times of the four variants of MirkDC in experiment #4 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures D.39 and D.43.

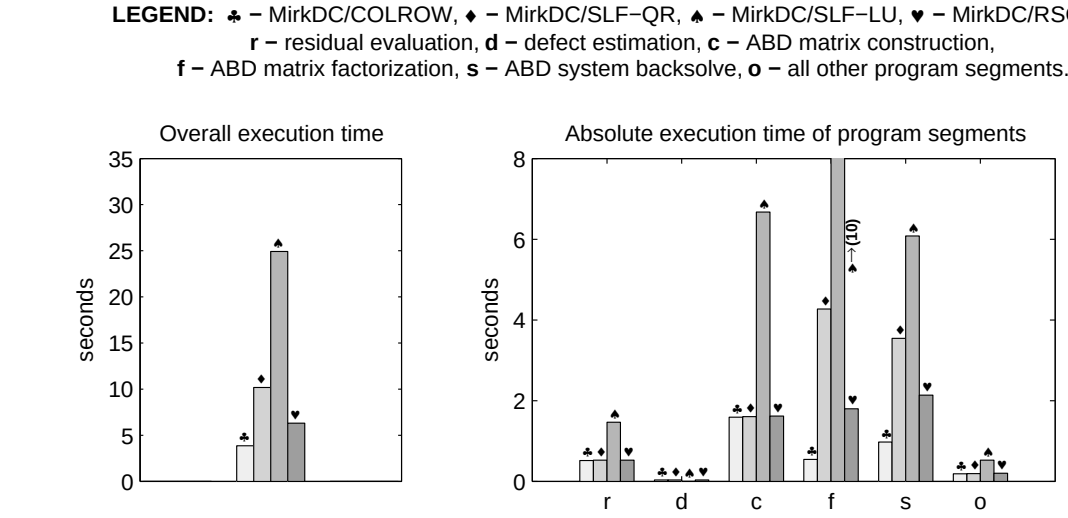


Figure D.43: Profiles of MirkDC/*SLF*-LU in experiment #4 of Table 4.11. This variant of MirkDC does not converge in this experiment (i.e. τ_{defect} is not satisfied). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 640, 1280, 2560, 5120 and 10240. This instability is investigated further in Figure D.44.

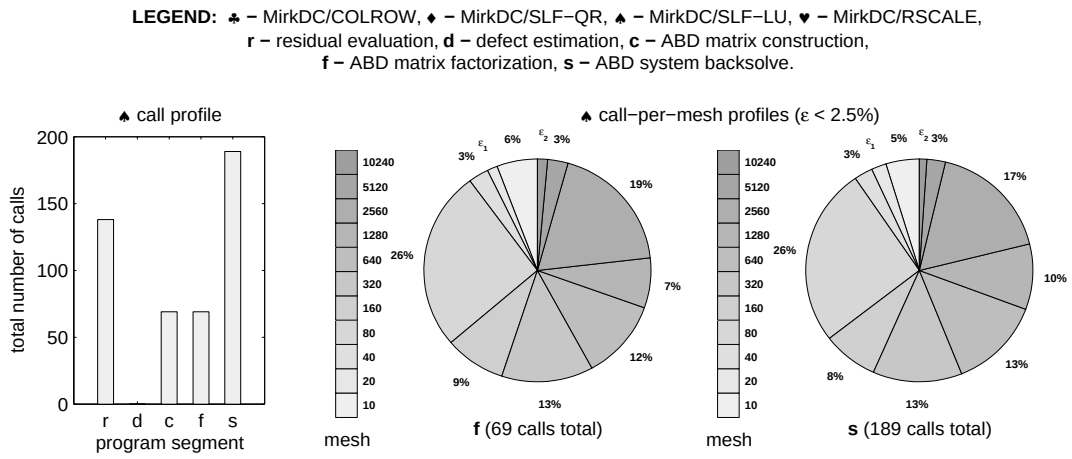


Figure D.44: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #4 (Figure D.43). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 8@mesh 640, 5@mesh 1280, 3-4@mesh 2560 and 1-2@mesh 5120.

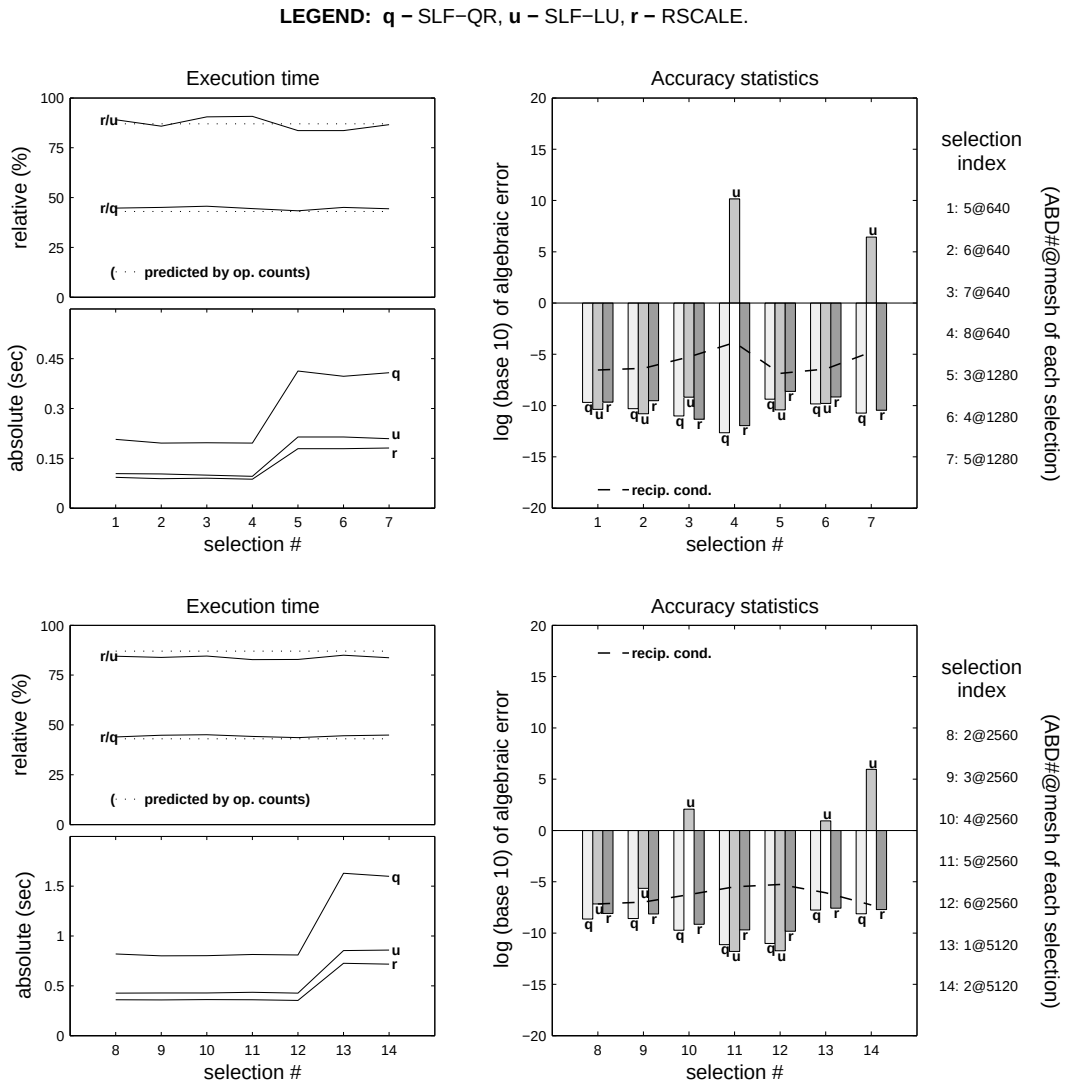


Figure D.45: Overall and selected program segment execution times of the four variants of MirkDC in experiment #5 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures D.39 and D.46.

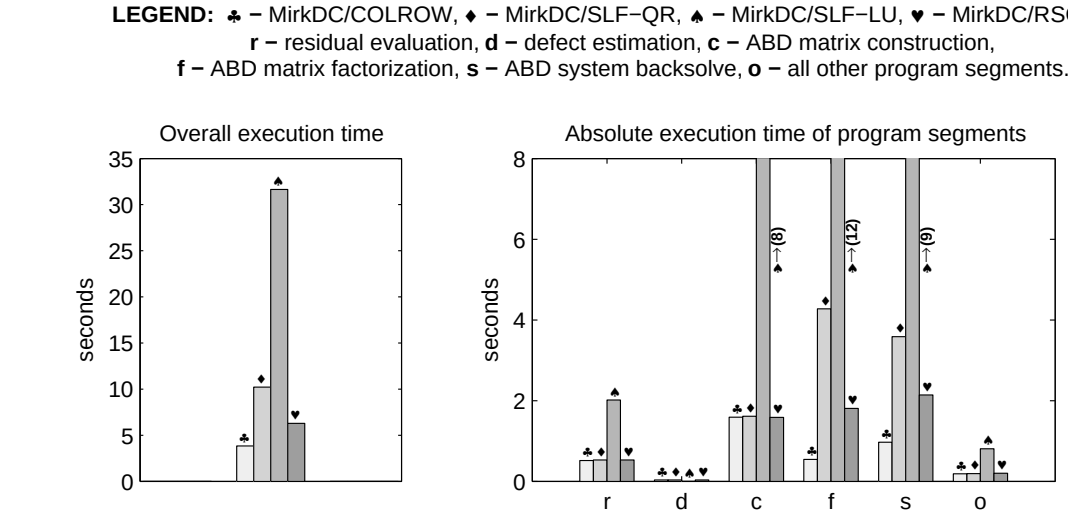


Figure D.46: Profiles of MirkDC/*SLF*-LU in experiment #5 of Table 4.11. This variant of MirkDC does not converge in this experiment (i.e. τ_{defect} is not satisfied). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 640, 1280, 2560, 5120 and 10240. This instability is investigated further in Figure D.47.

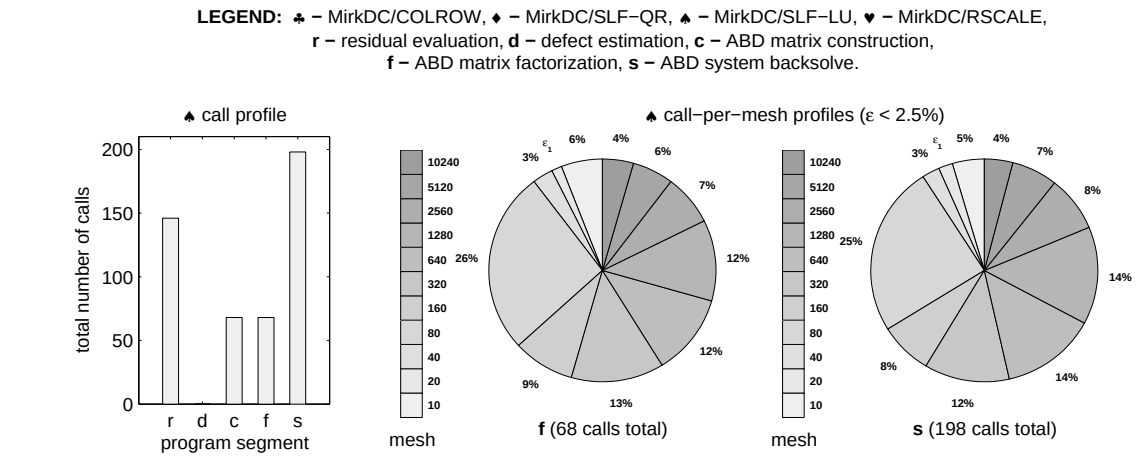


Figure D.47: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #5 (Figure D.46). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 8@mesh 640, 6-8@mesh 1280, 5@mesh 2560 and 4@mesh 5120.

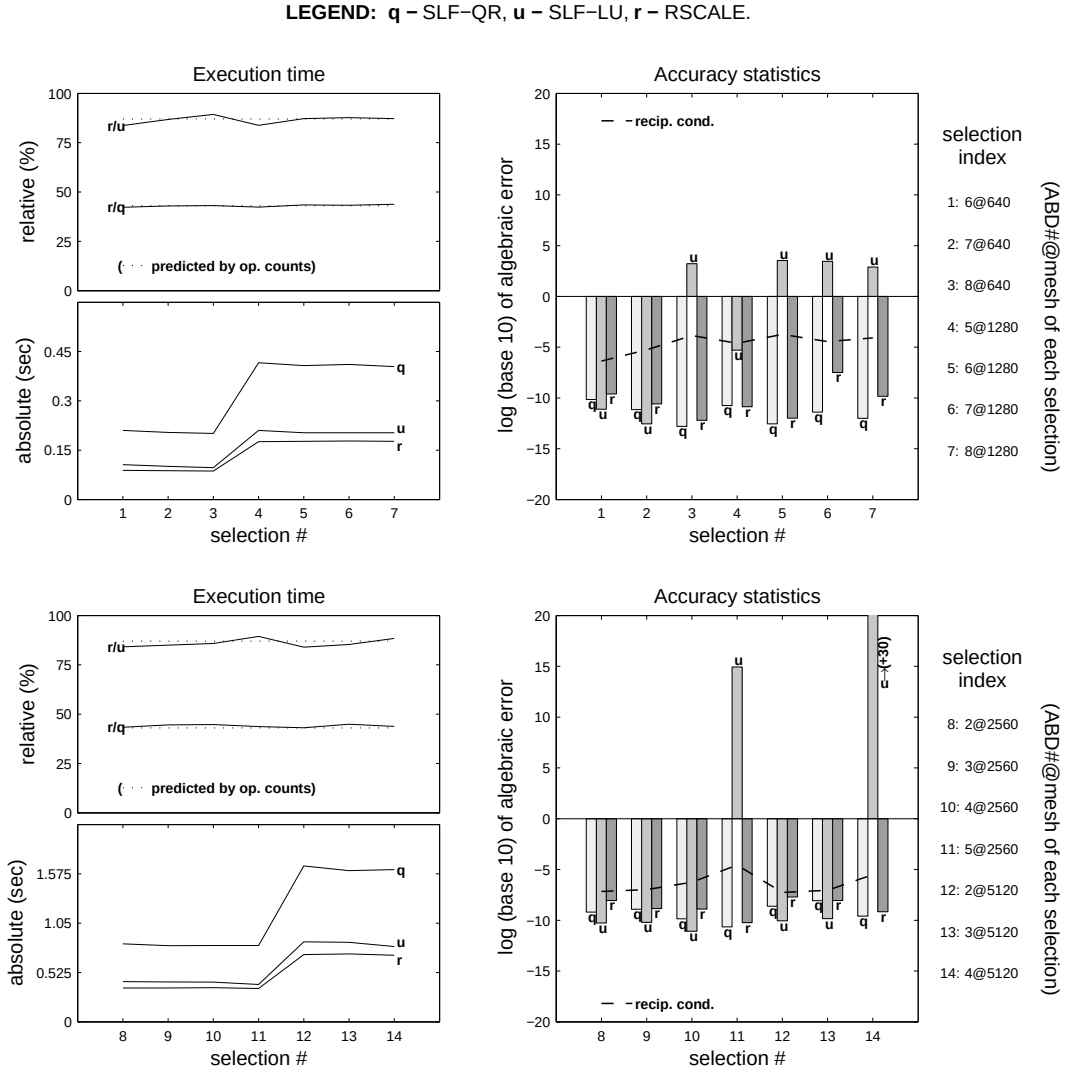


Figure D.48: Overall and selected program segment execution times of the four variants of MirkDC in experiment #6 of Table 4.11. This is the only experiment in Table 4.11 where *SLF*-LU does not exhibit instability. MirkDC/*SLF*-LU performance is comparable to the other variants of MirkDC, given the relative speeds of the parallel ABD system solvers.

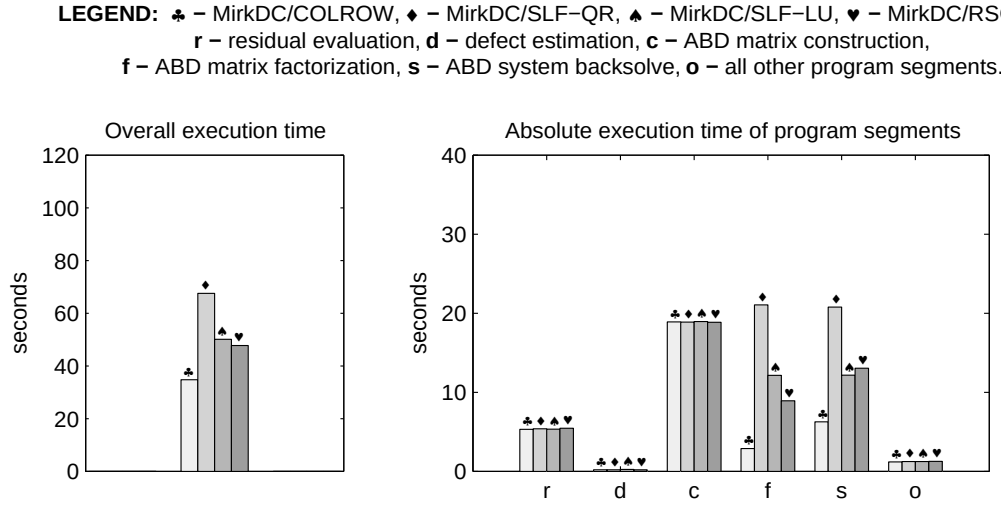


Figure D.49: Profiles of the four variants of MirkDC in experiment #6 of Table 4.11. Each variant converges in this experiment. Profiles are identical among variants.

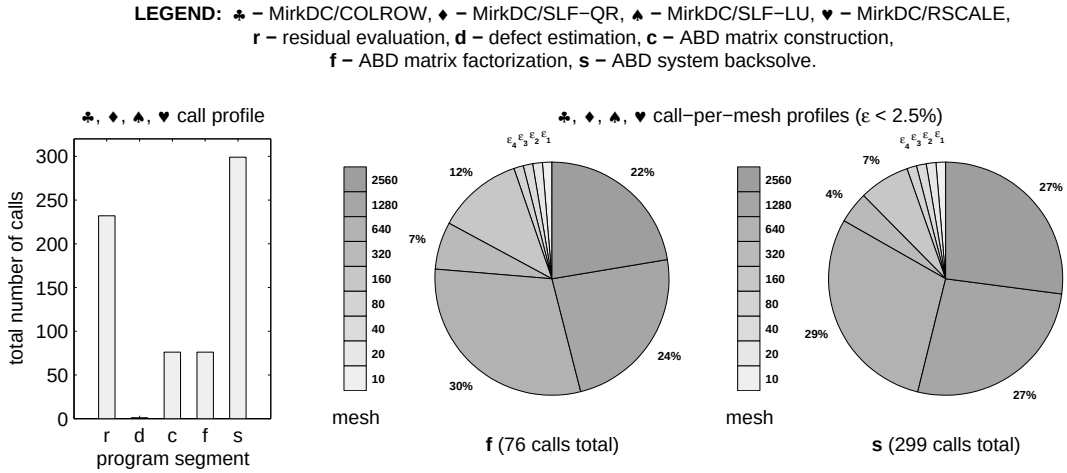


Figure D.50: Overall and selected program segment execution times of the four variants of MirkDC in experiment #7 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures D.51 and D.52.

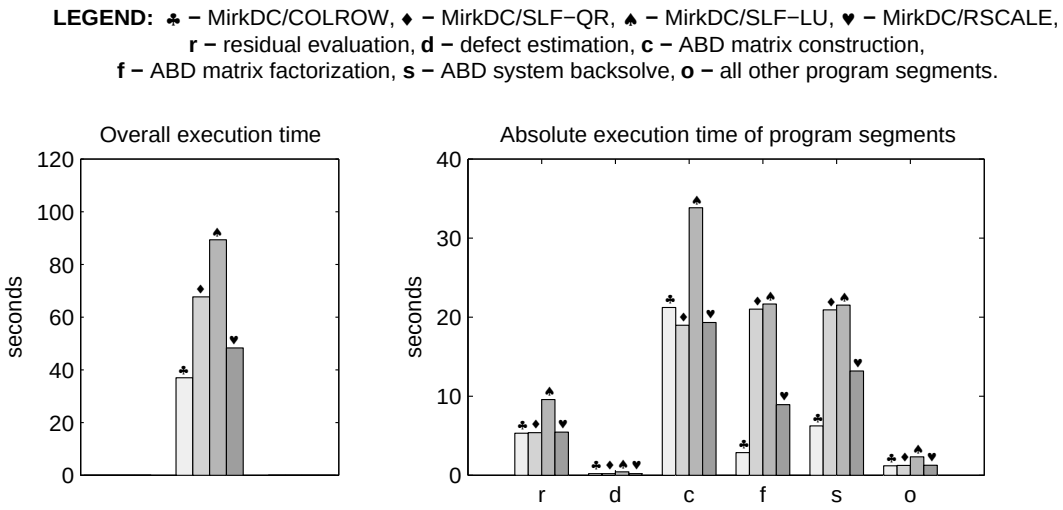


Figure D.51: Profiles of the *stable* variants of MirkDC (MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/RSCALE) in experiments #7 and #8 of Table 4.11. Each of these variants converges in both experiments. Profiles are identical among variants and between experiments.

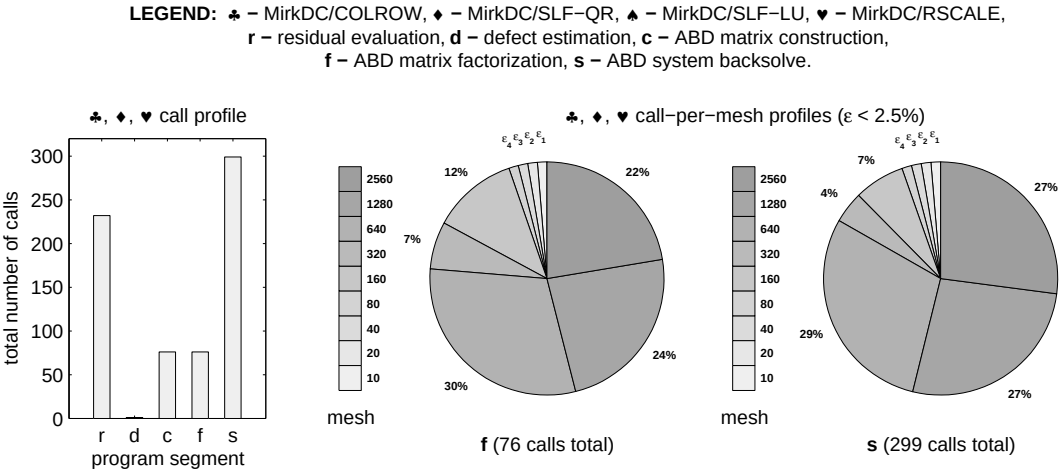


Figure D.52: Profiles of MirkDC/*SLF*-LU in experiment #7 of Table 4.11. This variant of MirkDC converges in this experiment, albeit less efficiently than the other three variants (Figure D.51). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 640, 1280, 2560 and 5120. This instability is investigated further in Figure D.53.

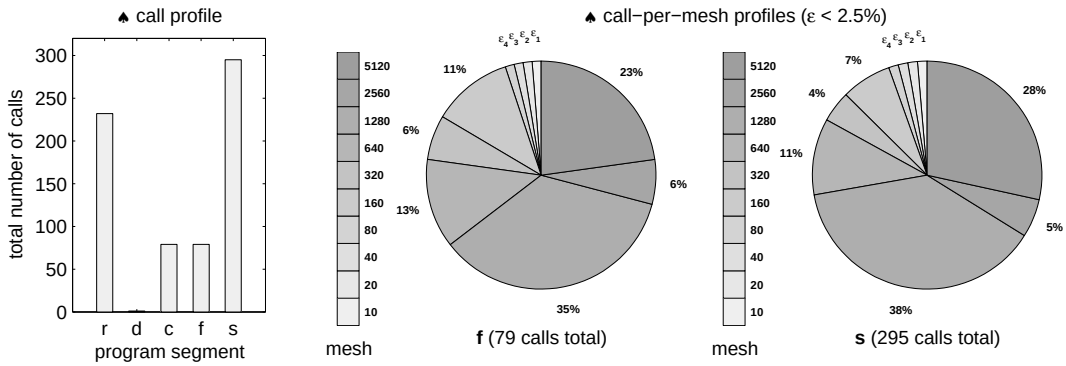


Figure D.53: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #7 (Figure D.52). Each extracted ABD matrix is well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 10@mesh 640, 5@mesh 1280, 4-5@mesh 2560 and 2@mesh 5120.

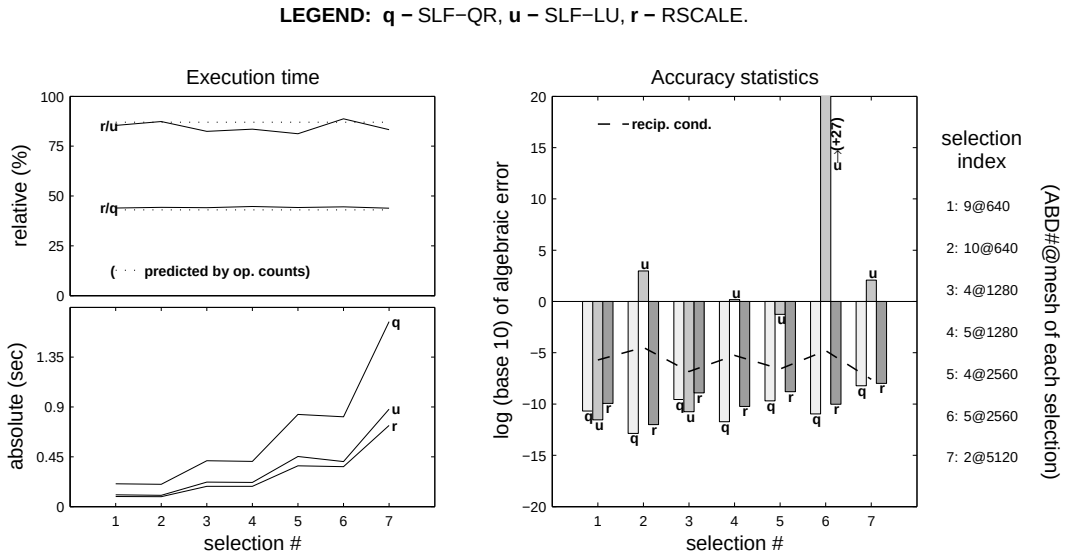


Figure D.54: Overall and selected program segment execution times of the four variants of MirkDC in experiment #8 of Table 4.11. The poor performance of MirkDC/*SLF*-LU relative to the other variants of MirkDC is explained by comparing the subroutine call and call-per-mesh profiles in Figures D.51 and D.55.

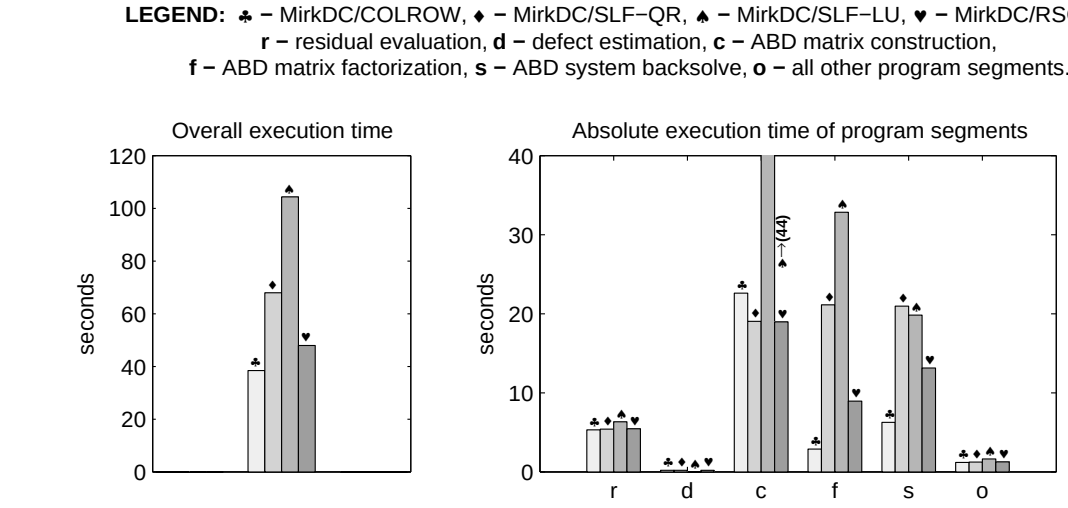


Figure D.55: Profiles of MirkDC/*SLF*-LU in experiment #8 of Table 4.11. This variant of MirkDC does not converge in this experiment (i.e. τ_{defect} is not satisfied). Instability was detected in the *SLF*-LU factorization of one or more ABD systems built on meshes of size 1280, 2560, 5120 and 10240. This instability is investigated further in Figure D.56.

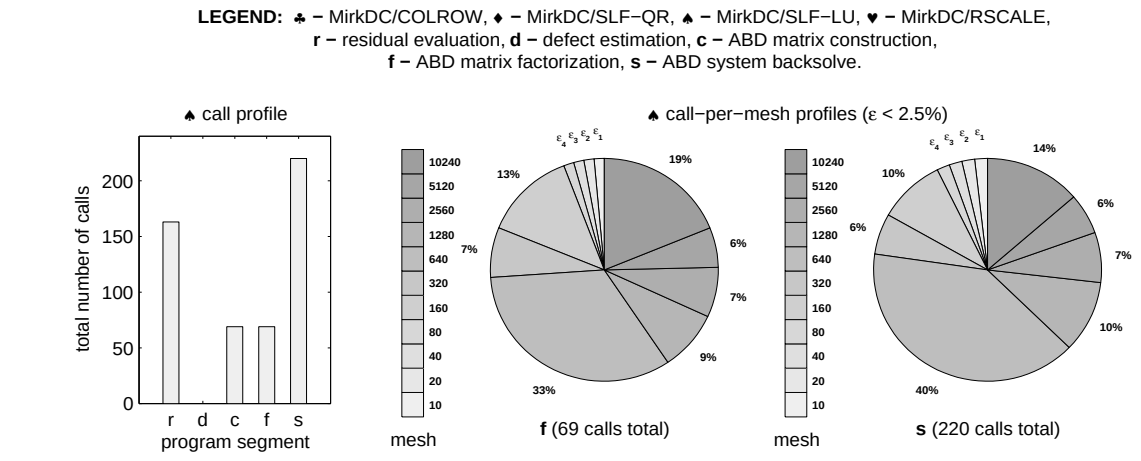


Figure D.56: Execution time and accuracy of the three parallel ABD solvers when solving selected systems extracted from the MirkDC/*SLF*-LU solution in experiment #8 (Figure D.55). Although extracted ABD matrices 20-23@mesh 640 are poorly conditioned, each of the ABD solvers computes a reasonably accurate solution to the respective systems. Extracted ABD matrices built on meshes of size 1280, 2560, and 5120 are comparitively well-conditioned. Of the three ABD solvers, only *SLF*-LU exhibits instability when solving ABD systems 6@mesh 1280, 5@mesh 2560 and 4@mesh 5120.

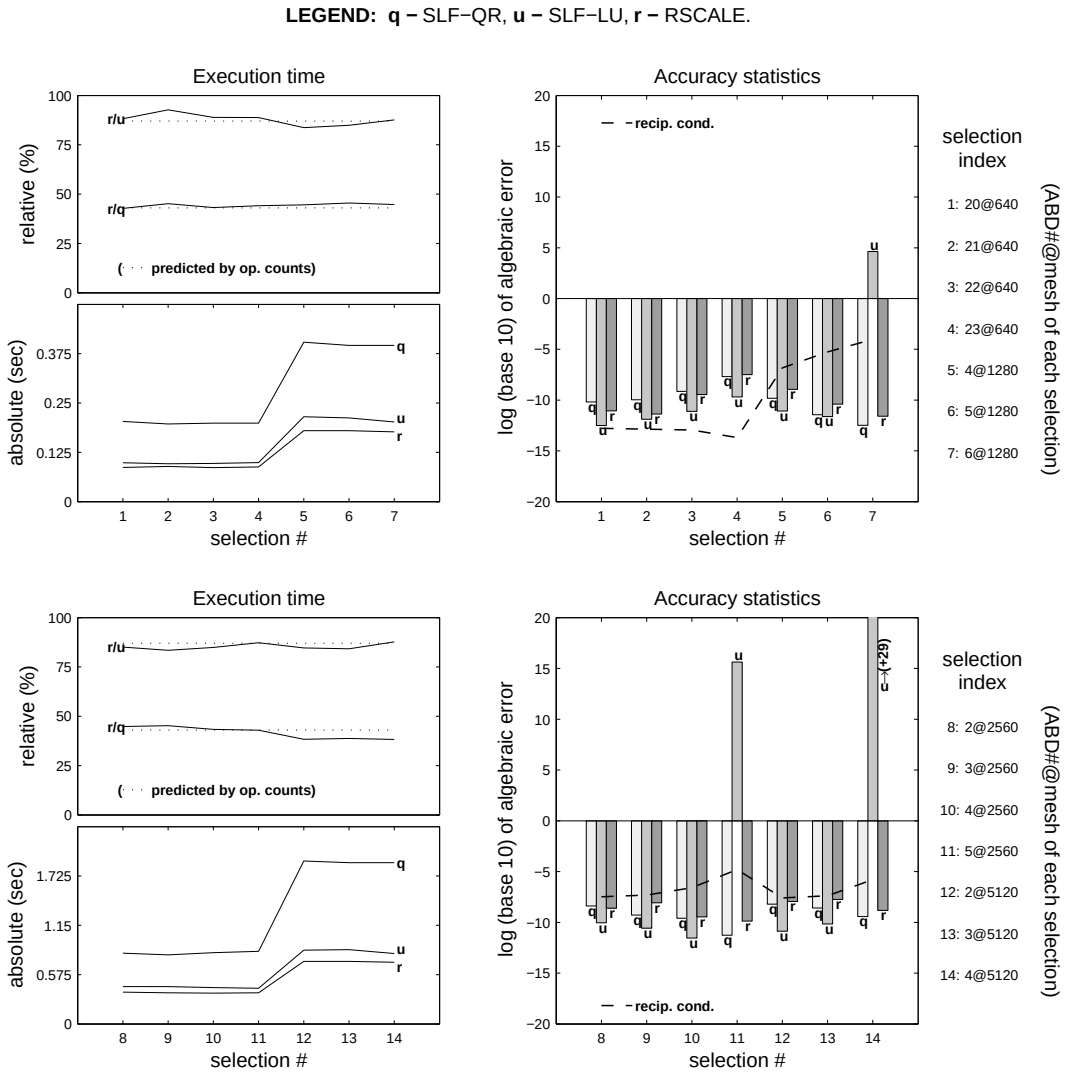


Figure D.57: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 5@mesh 2560 extracted from the MirkDC/*SLF*-LU solution in experiment #5 (Figure D.46). Of the three ABD solvers, only *SLF*-LU exhibits instability when using 2-8 partitions. Surprisingly, non-partitioned *SLF*-LU is *stable* on this problem.

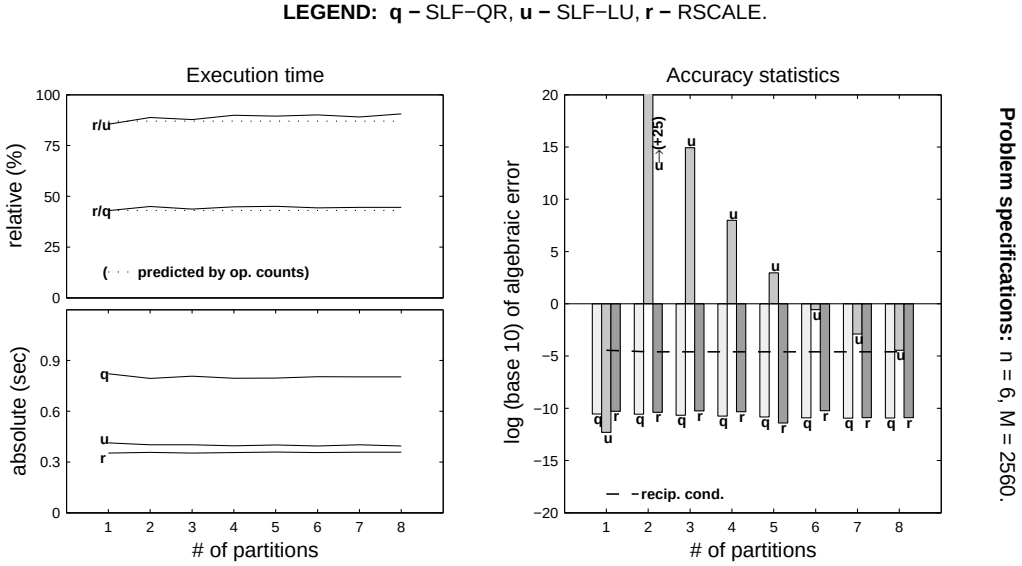


Figure D.58: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 4@mesh 5120 extracted from the MirkDC/*SLF*-LU solution in experiment #5 (Figure D.46). Of the three ABD solvers, only *SLF*-LU exhibits instability when using 2-8 partitions. Surprisingly, non-partitioned *SLF*-LU is *stable* on this problem.

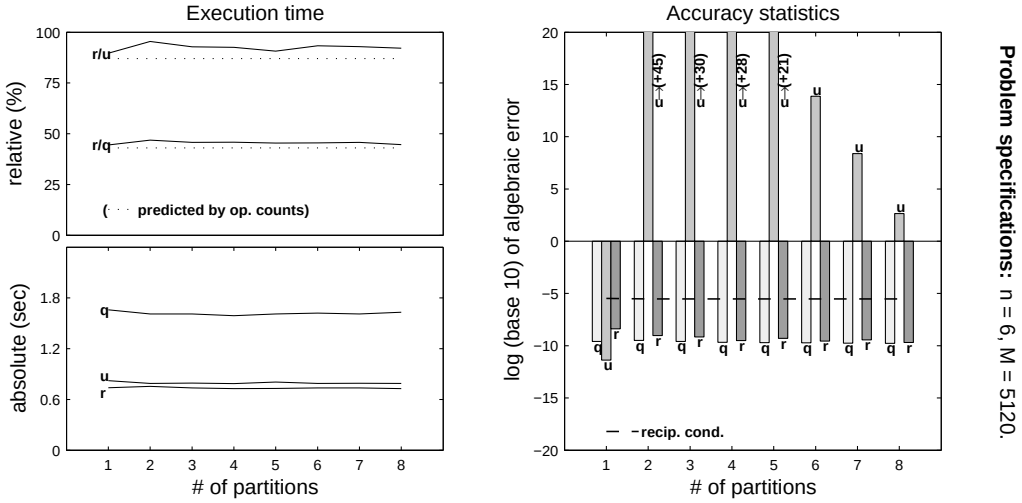


Figure D.59: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 10@mesh 640 extracted from the MirkDC/*SLF*-LU solution in experiment #7 (Figure D.52). Of the three ABD solvers, only *SLF*-LU exhibits instability when using 2-8 partitions. Surprisingly, non-partitioned *SLF*-LU is *stable* on this problem.

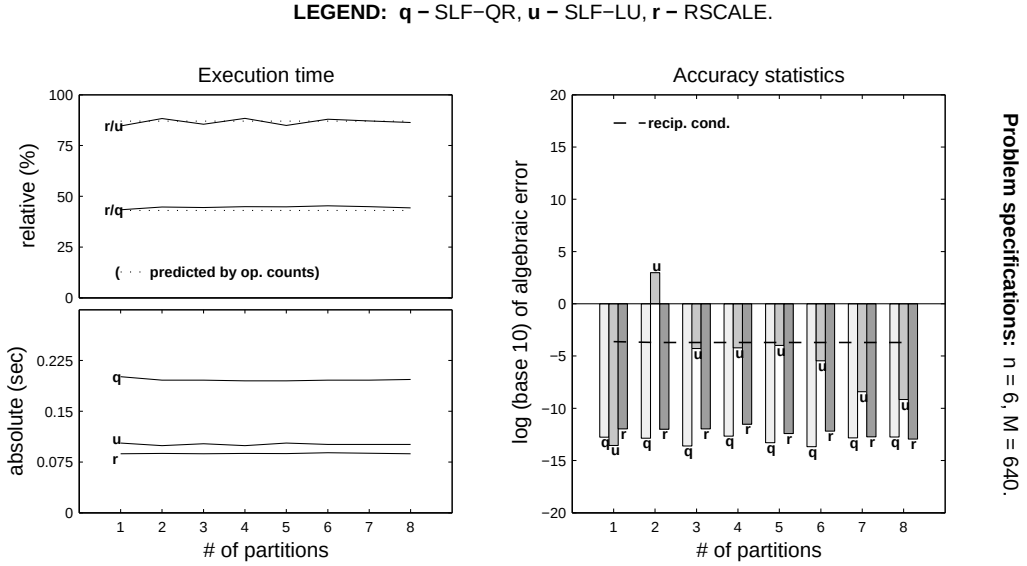


Figure D.60: Execution time and accuracy of the three parallel ABD solvers when solving, using 1-8 partitions, ABD system 6@mesh 1280 extracted from the MirkDC/*SLF*-LU solution in experiment #8 (Figure D.55). Of the three ABD solvers, only *SLF*-LU exhibits instability when using 2-8 partitions. Surprisingly, non-partitioned *SLF*-LU is *stable* on this problem.

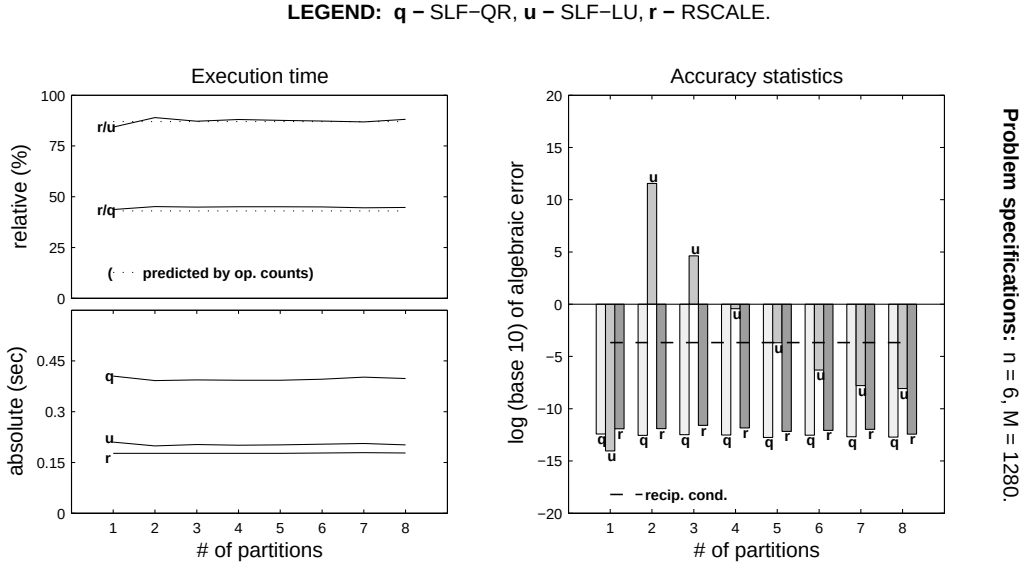


Table D.5: Numerical results index for experiments #3-#8 of Table 4.12 in §4.4.4.

exp. #	Results are shown	
	in Figure(s)	on page(s)
3	D.61, D.62, D.63, D.64, D.65	237, 238, 239
4	D.66, D.67, D.68, D.69, D.70	240, 241, 242
5	D.71, D.72, D.73	243, 244
6	D.74, D.75, D.76, D.77	245, 246
7	D.78, D.79, D.80	247, 248
8	D.81, D.82, D.83, D.84, D.85	249, 250, 251

D.4 Problems Where Sequential MirkDC/RSCALE Outperforms MirkDC/COLROW

Table 4.12 in §4.4.4 lists eight experiments on difficult SWF-III problems. In each experiment, the convergence pattern of at least one variant of MirkDC differs from the others. This appendix contains numerical results for experiments #3-#8 of that table. Figures summarizing the output of each experiment are indexed in Table D.5.

The output of an experiment is summarized in a figure showing the overall and selected program segment execution times of each variant of MirkDC, and two or more figures showing the subroutine call and call-per-mesh profiles for each distinct convergence pattern. We note in the figure caption when two or more variants share the same pattern, or when a pattern does not lead to convergence.

Convergence is achieved by each variant in each experiment, except for MirkDC/*SLF*-LU in experiment #4. In all but experiments #5 and #8, MirkDC/RSCALE converges faster than the other variants, including MirkDC/COLROW. In experiments #5 and #8 MirkDC/COLROW is marginally faster, even though the MirkDC/COLROW convergence sequence shows one or more larger ABD systems than generated by MirkDC/RSCALE. (Compare profiles in Figures D.72 and D.73, and Figures D.82 and D.85.)

Figure D.61: Overall and selected program segment execution times of the four variants of MirkDC in experiment #3 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments. Compare the subroutine call and call-per-mesh profiles in Figures D.62, D.63, D.64 and D.65.

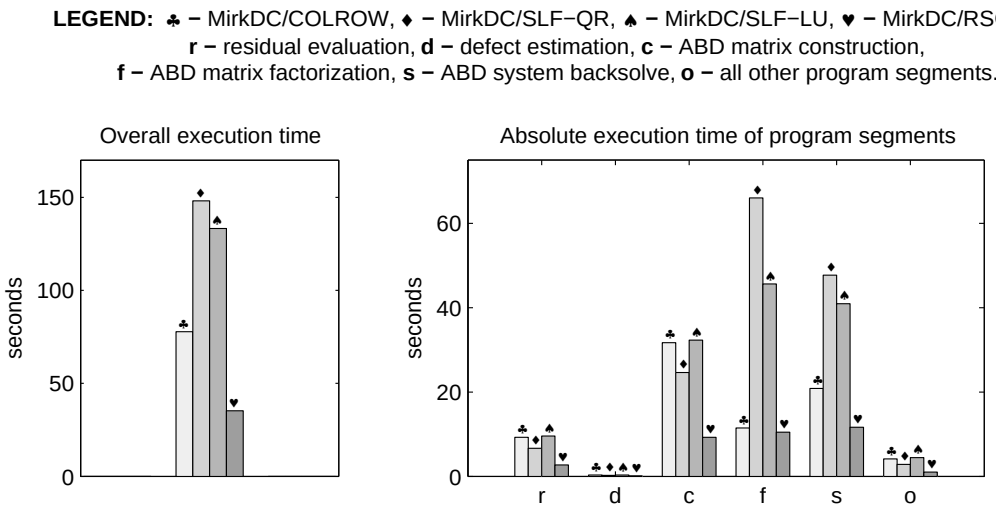


Figure D.62: Profiles of MirkDC/COLROW in experiment #3 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.65).

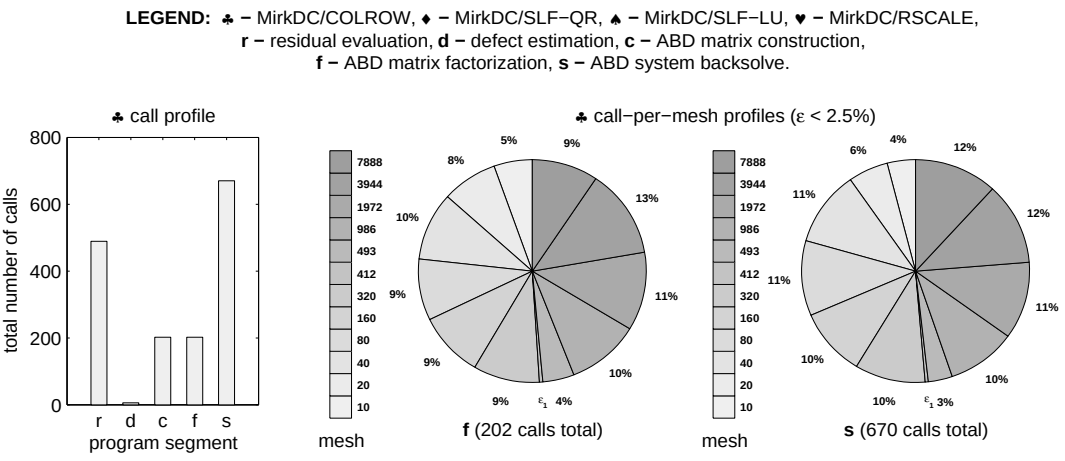


Figure D.63: Profiles of MirkDC/*SLF*-QR in experiment #3 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.65).

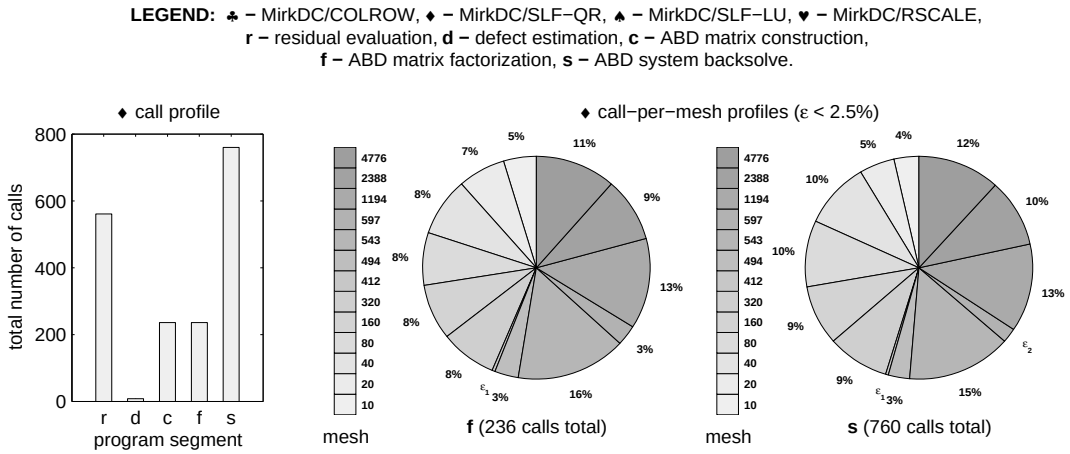


Figure D.64: Profiles of MirkDC/*SLF*-LU in experiment #3 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.65).

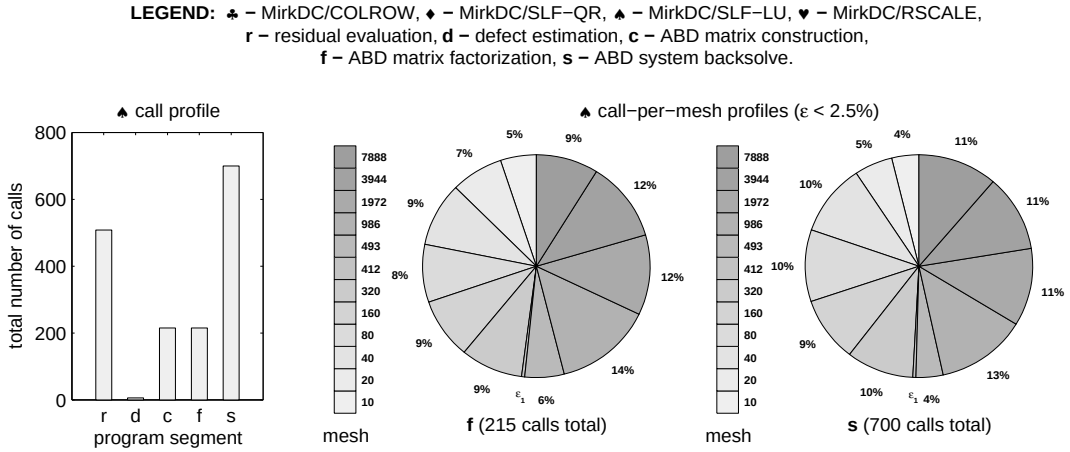


Figure D.65: Profiles of MirkDC/RSCALE in experiment #3 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 2180. Compare these profiles to those of the other variants shown in Figures D.62, D.63 and D.64.

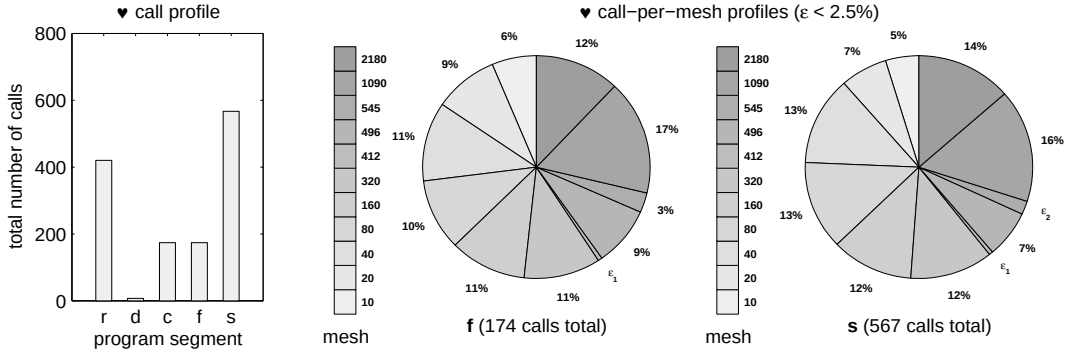


Figure D.66: Overall and selected program segment execution times of the four variants of MirkDC in experiment #4 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments. Compare the subroutine call and call-per-mesh profiles in Figures D.67, D.68, D.69 and D.70. Note that MirkDC/*SLF*-LU does not converge in this experiment.

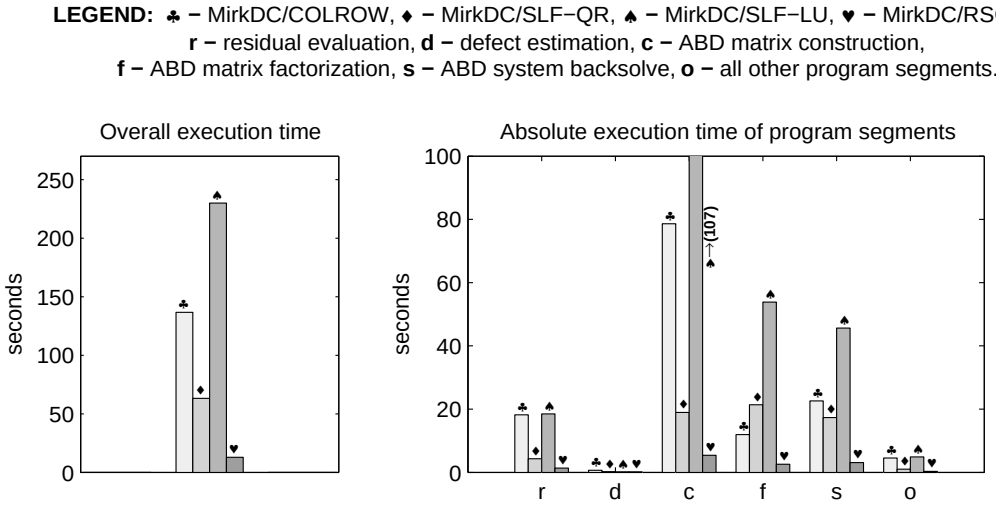


Figure D.67: Profiles of MirkDC/COLROW in experiment #4 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.70).

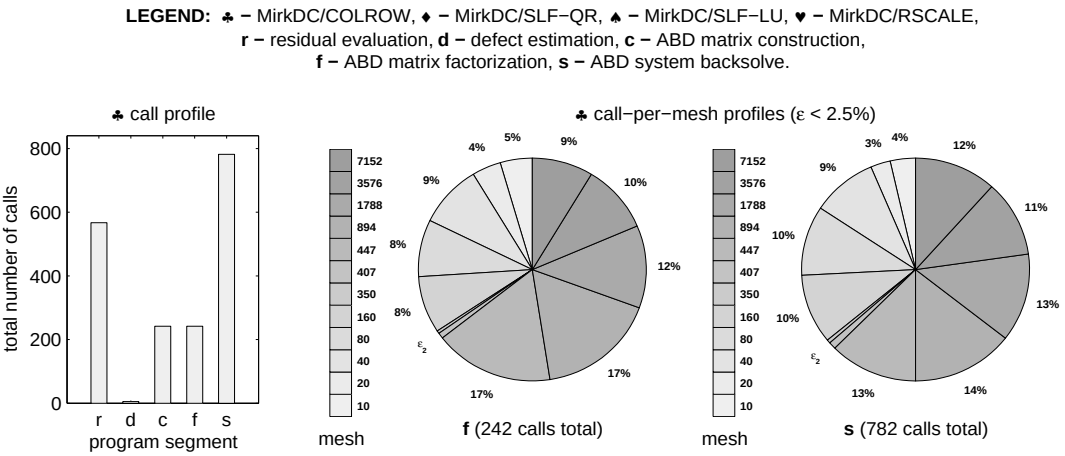


Figure D.68: Profiles of MirkDC/*SLF*-QR in experiment #4 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.70).

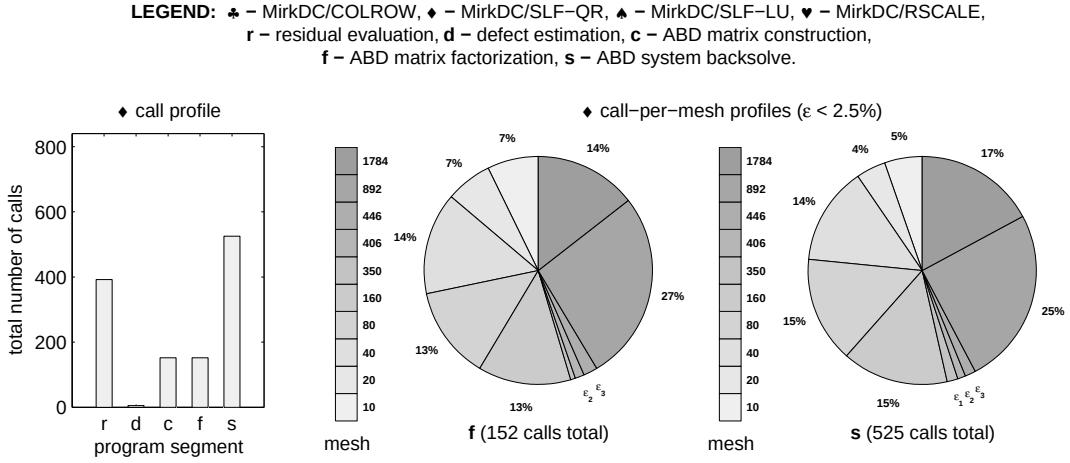


Figure D.69: Profiles of MirkDC/*SLF*-LU in experiment #4 of Table 4.12. This variant of MirkDC *does not* converge in this experiment (i.e. τ_{defect} is not satisfied).

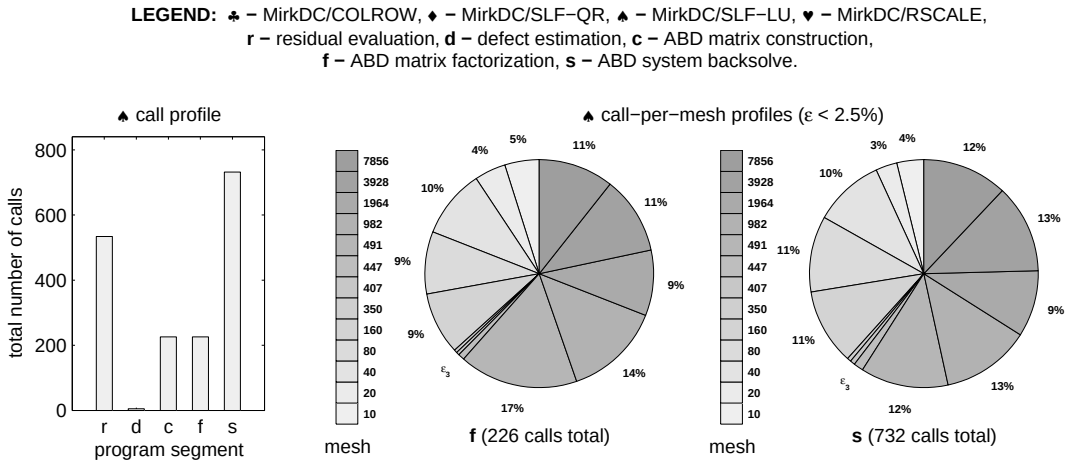


Figure D.70: Profiles of MirkDC/RSCALE in experiment #4 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 502. Compare these profiles to those of the other variants shown in Figures D.67, D.68 and D.69.

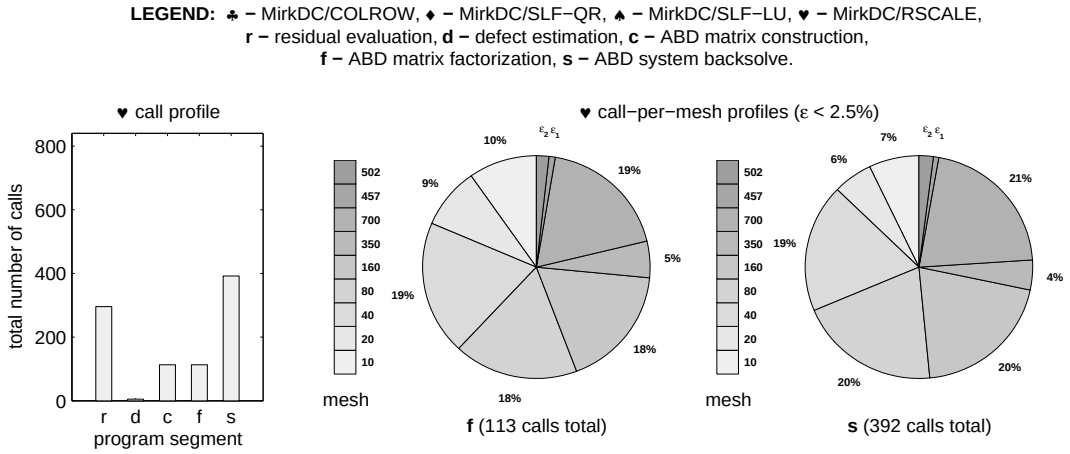


Figure D.71: Overall and selected program segment execution times of the four variants of MirkDC in experiment #5 of Table 4.12. MirkDC/COLROW is marginally faster than MirkDC/RSCALE with respect to overall execution time, however it generates larger ABD systems. Compare the subroutine call and call-per-mesh profiles in Figures D.72 and D.73.

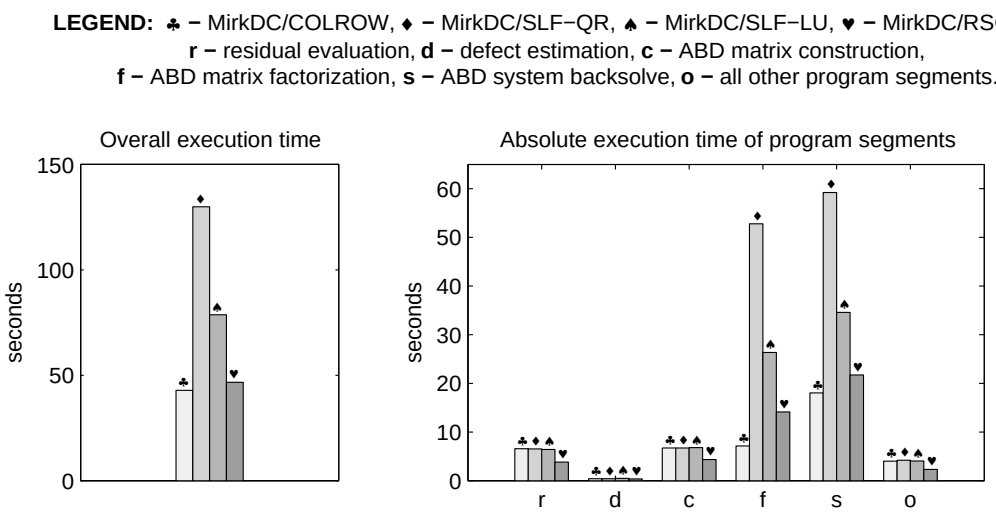


Figure D.72: Profiles of MirkDC/COLROW, MirkDC/*SLF*-QR and MirkDC/*SLF*-LU in experiment #5 of Table 4.12. These three variants of MirkDC converge in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.73).

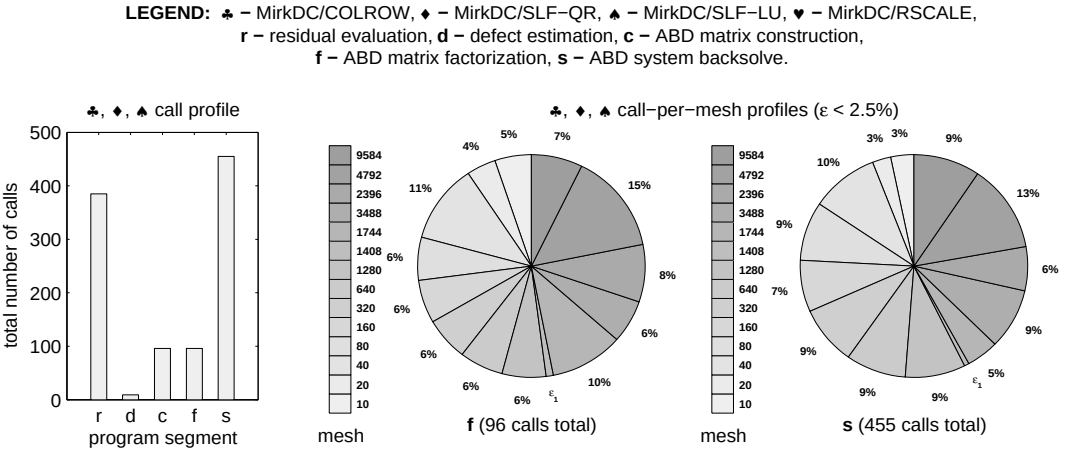


Figure D.73: Profiles of MirkDC/RSCALE in experiment #5 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 4792. Compare these profiles to those of the other variants shown in Figure D.72.

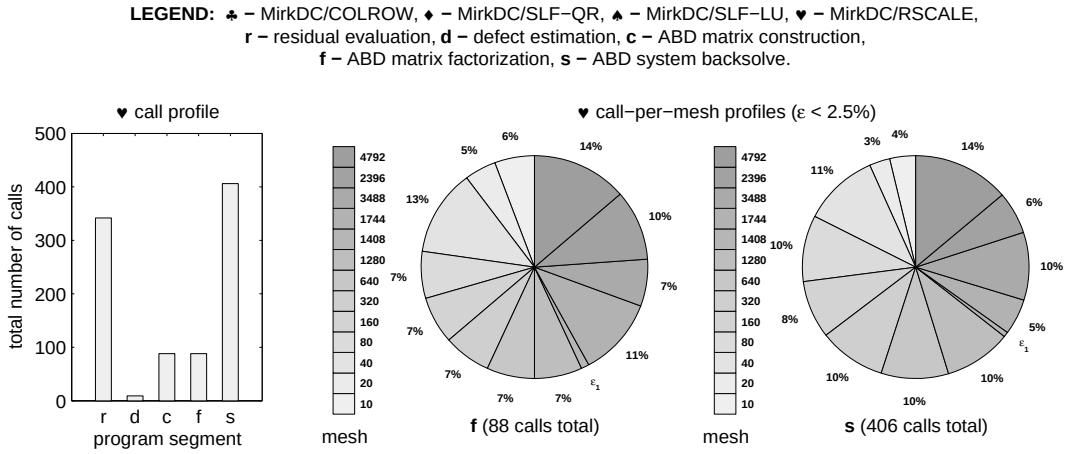


Figure D.74: Overall and selected program segment execution times of the four variants of MirkDC in experiment #6 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments. Compare the subroutine call and call-per-mesh profiles in Figures D.75, D.76 and D.77.

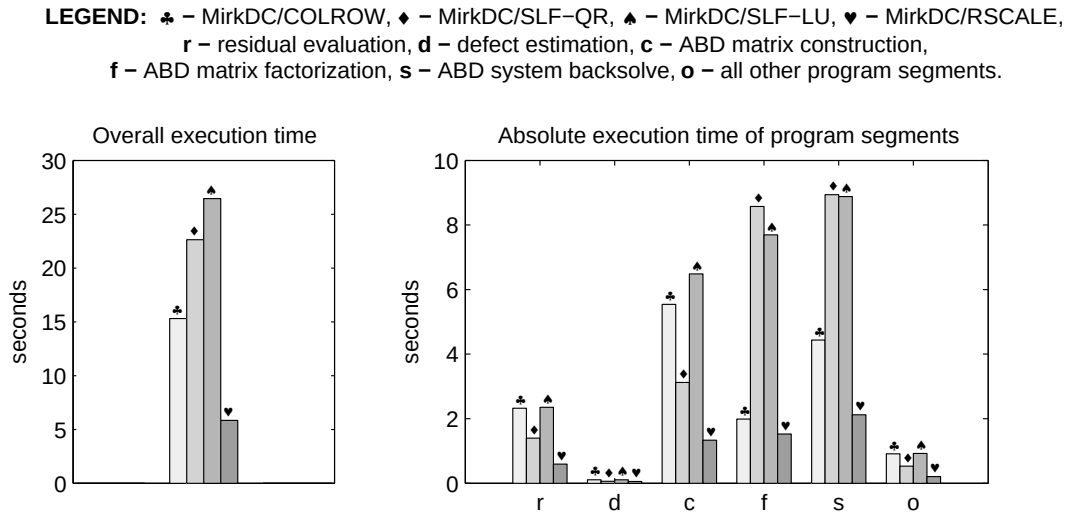


Figure D.75: Profiles of MirkDC/COLROW and MirkDC/*SLF*-LU in experiment #6 of Table 4.12. These two variants of MirkDC converge in this experiment, but less efficiently than either MirkDC/RSCALE or MirkDC/*SLF*-QR (Figures D.77 and D.76).

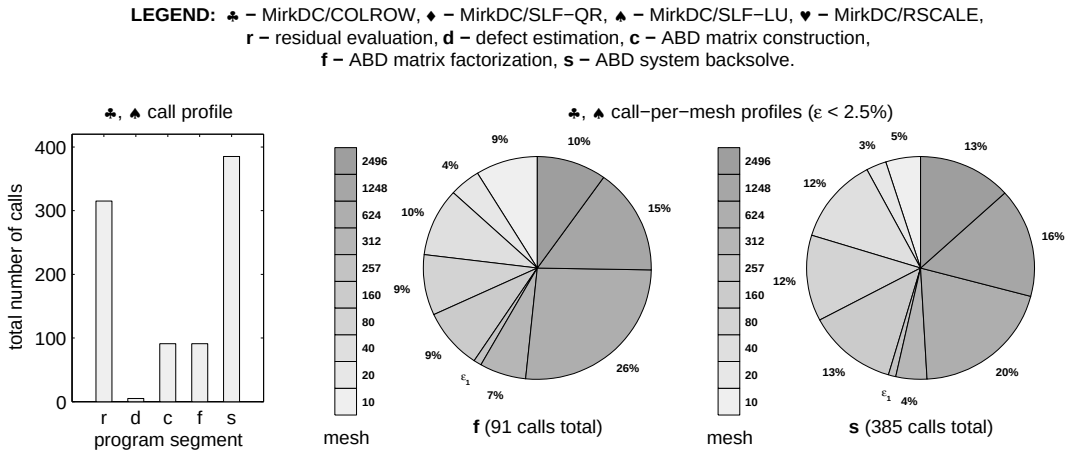


Figure D.76: Profiles of MirkDC/*SLF*-QR in experiment #6 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.77).

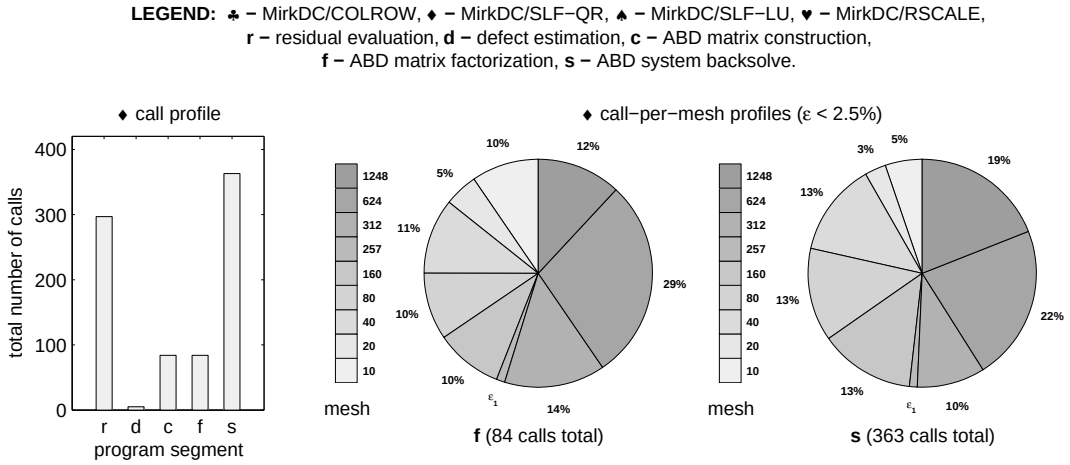


Figure D.77: Profiles of MirkDC/RSCALE in experiment #6 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 686. Compare these profiles to those of the other variants shown in Figures D.75 and D.76.

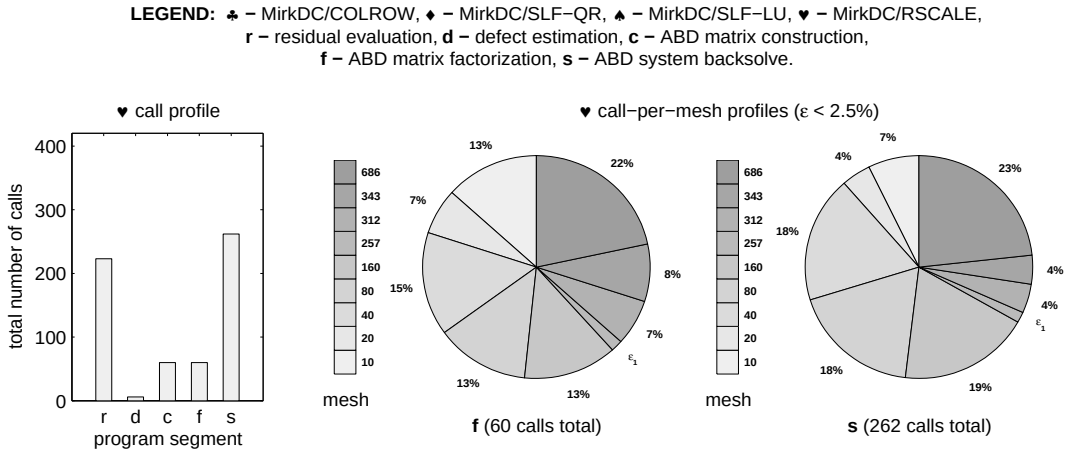


Figure D.78: Overall and selected program segment execution times of the four variants of MirkDC in experiment #7 of Table 4.12. MirkDC/RSCALE outperforms MirkDC/COLROW in all program segments. Compare the subroutine call and call-per-mesh profiles in Figures D.79 and D.80.

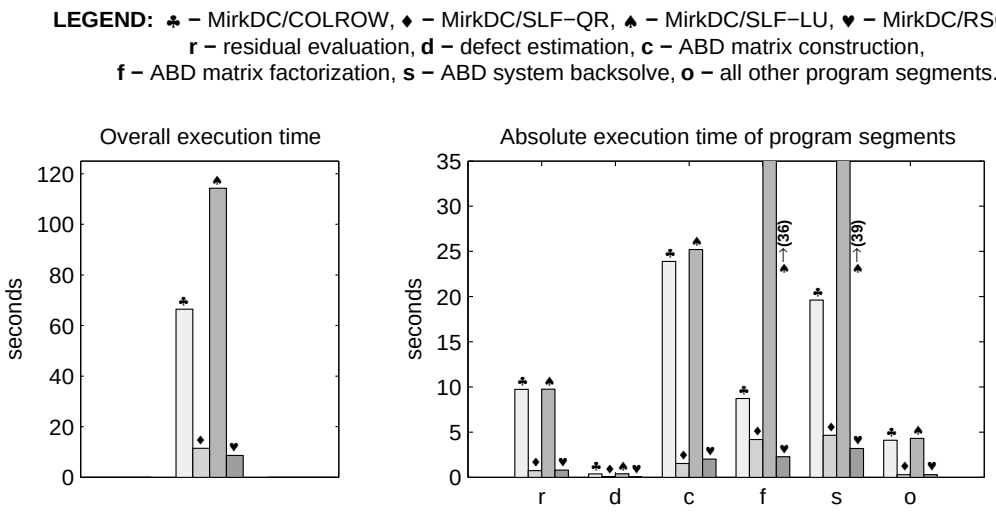


Figure D.79: Profiles of MirkDC/COLROW in experiment #7 of Table 4.12. Profiles of MirkDC/*SLF*-LU differ only slightly. These two variants of MirkDC converge in this experiment, but less efficiently than either MirkDC/RSCALE or MirkDC/*SLF*-QR (Figure D.80).

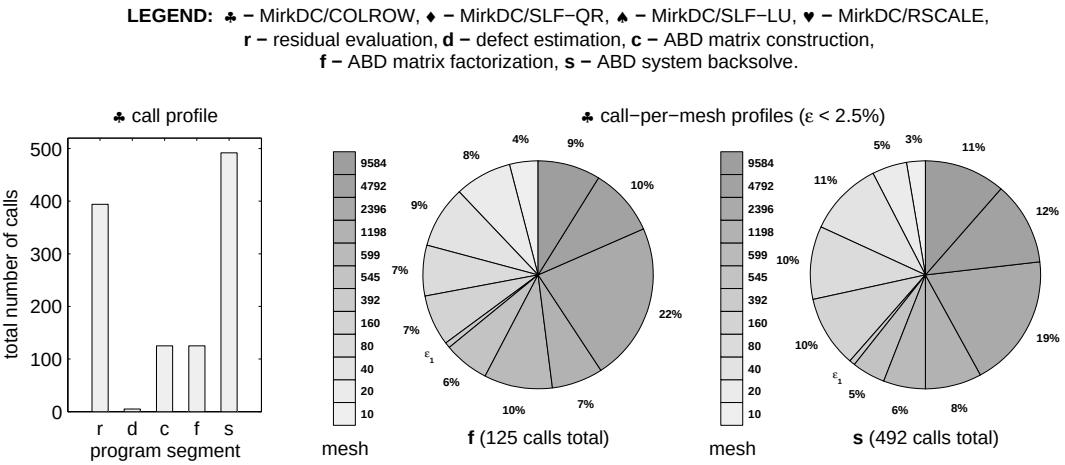


Figure D.80: Profiles of MirkDC/RSCALE in experiment #7 of Table 4.12. Profiles of MirkDC/*SLF*-QR differ only slightly. These two variants of MirkDC converge in this experiment, with a final mesh of size 1092. Compare these profiles to those of the other variants shown in Figure D.79.

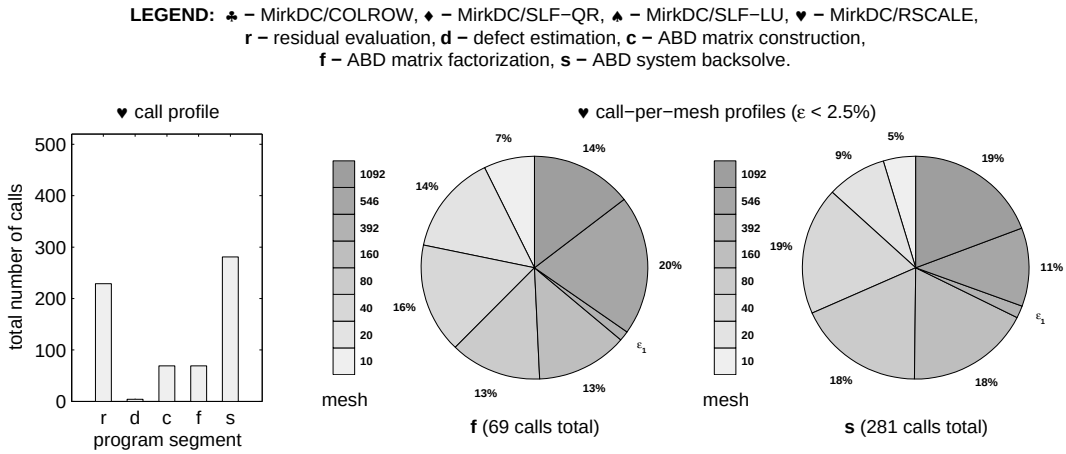


Figure D.81: Overall and selected program segment execution times of the four variants of MirkDC in experiment #8 of Table 4.12. MirkDC/COLROW is marginally faster than MirkDC/RSCALE with respect to overall execution time, however it generates larger ABD systems. Compare the subroutine call and call-per-mesh profiles in Figures D.82 D.83, D.84 and D.85.

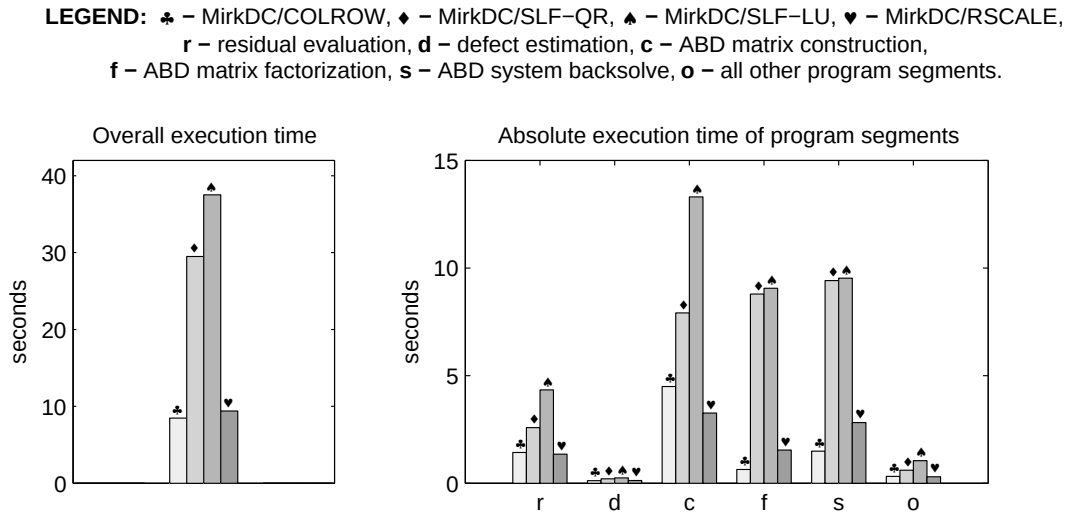


Figure D.82: Profiles of MirkDC/COLROW in experiment #8 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.85).

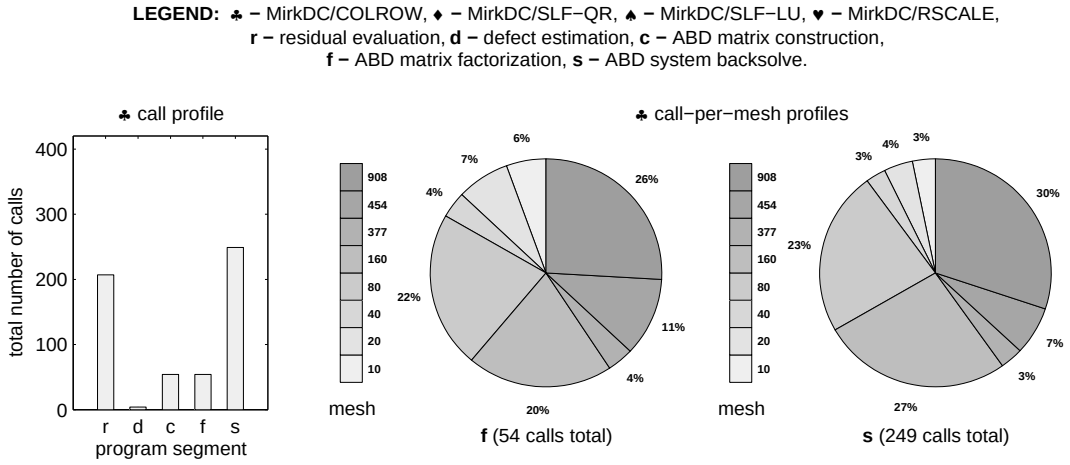


Figure D.83: Profiles of MirkDC/*SLF*-QR in experiment #8 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.85).

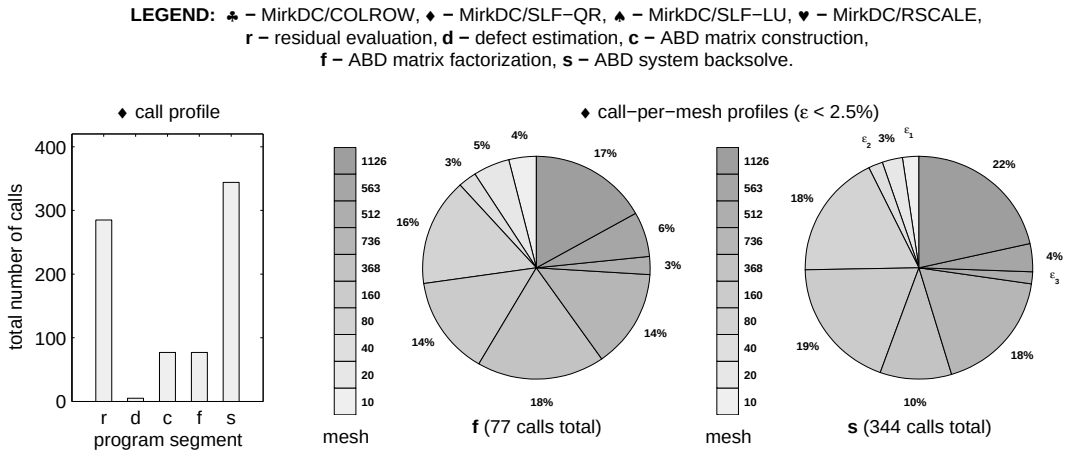


Figure D.84: Profiles of MirkDC/*SLF*-LU in experiment #8 of Table 4.12. This variant of MirkDC converges in this experiment, but less efficiently than MirkDC/RSCALE (Figure D.85).

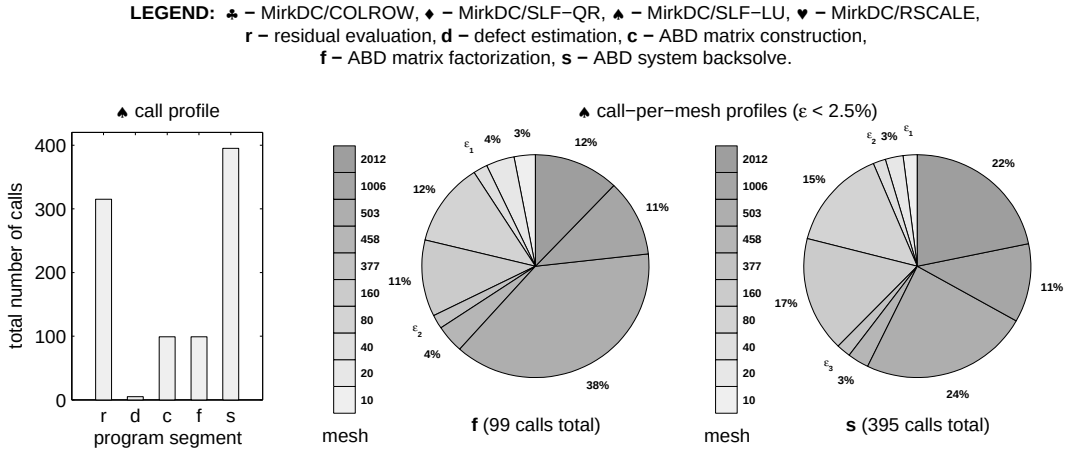
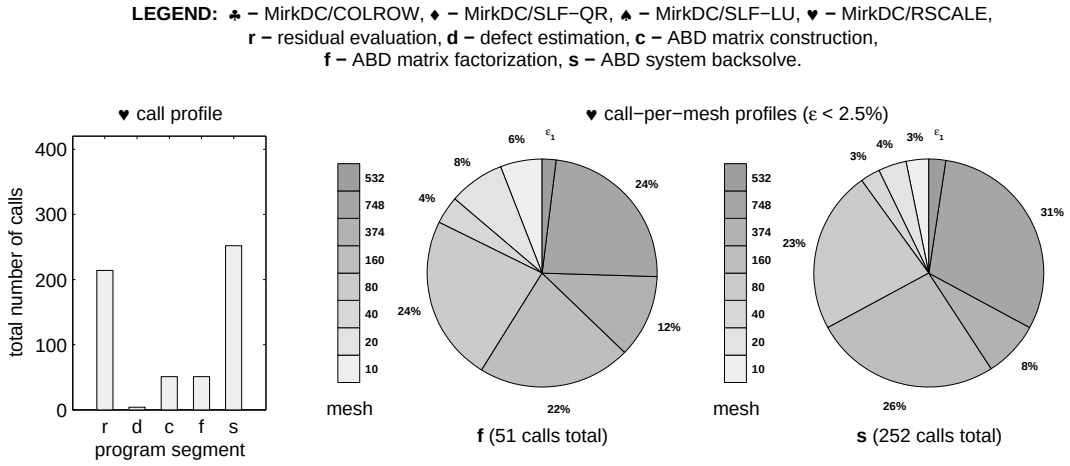


Figure D.85: Profiles of MirkDC/RSCALE in experiment #8 of Table 4.12. This variant of MirkDC converges in this experiment, with a final mesh of size 532. Compare these profiles to those of the other variants shown in Figures D.82, D.83 and D.84.



Appendix E

Fortran Source Listings

E.1 RSCALE

```
      subroutine rscale (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, pivot, iflag, nparts, work)
c
      double precision lftblk(1), array(1), rgtblk(1), b(1), work(1)
      integer nrwblk, nbloks, pivot(1), iflag, nparts
c     ***------------------------------------------------------------***
c     *   This subroutine solves the linear system A x = b where     *
c     *   A is an Almost Block Diagonal matrix of the form           *
c     *                                                              *
c     *             lftblk                    rgtblk               *
c     *             array(,,1)                                     *
c     *                  array(,,2)                                *
c     *                       .                                    *
c     *                         .                                  *
c     *                           .                                *
c     *                              array(,,nbloks)               *
c     *                                                            *
c     *   lftblk and rgtblk are each nrwblkxnrwblk, array(,,k)     *
c     *   is nrwblkx2*nrwblk, and array(,,k) and array(,,k+1)      *
c     *   overlap by nrwblk columns.  The linear system is square  *
c     *   and of order (nbloks+1)*nrwblk.                          *
c     *                                                            *
c     *  [ Note:  ABDs often arise in other forms.  For example,   *
c     *       lftblk and rgtblk may be uncoupled so that lftblk    *
c     *       appears at the top of the matrix and rgtblk appears  *
c     *       at the bottom.  In this case, the ABD system first   *
c     *       can be transformed into the correct form for input   *
c     *       to 'rscale' using an auxiliary routine included      *
c     *       with this package.  See 'couple' for details.        *
c     *       Alternatively, 'rscalc', a modified version of       *
c     *       'rscale' that incorporates 'couple', can be used.   ]  *
c     *                                                            *
c     *   THE ALGORITHM:                                           *
c     *                                                            *
c     *   The system is decomposed and solved using a variant      *
c     *   of the parallel Rescaling algorithm described in [1].    *
```

```
c     *   Parallelism is achieved by slicing the system into        *
c     *   'nparts' partitions in such a way that each partition     *
c     *   can be processed independently.  Assuming at least one    *
c     *   processor is available per partition, a speed-up of S     *
c     *   (over sequential Rescaling) may be attained where         *
c     *                                                             *
c     *            1 <= S < nparts,                                 *
c     *                                                             *
c     *        with  S = 1       if  nbloks < 2*nparts,             *
c     *         and  S ~ nparts  if  nparts << nbloks/nparts.       *
c     *                                                             *
c     *   In other words, for systems of sufficiently high order,   *
c     *   speed-up is approximately linear with respect to nparts   *
c     *   when nparts is sufficiently small.  Sample problems and   *
c     *   timing benchmarks are included with this package.         *
c     *                                                             *
c     *   PARAMETERS:                                               *
c     *                                                             *
c     *    on entry                                                 *
c     *                                                             *
c     *      lftblk [double precision(nrwblk,nrwblk)]               *
c     *             The top left block of the ABD matrix.           *
c     *                                                             *
c     *       array [double precision(nrwblk,2*nrwblk,nbloks)]      *
c     *             array(,,k) contains the k-th nrwblkx2*nrwblk    *
c     *             block of the ABD matrix.                        *
c     *                                                             *
c     *      nrwblk [integer]                                       *
c     *             The number of rows in lftblk, array(,,k),       *
c     *             and rgtblk.  The number of columns in           *
c     *             lftblk and rgtblk.  There are 2*nrwblk          *
c     *             columns in array(,,k).                          *
c     *                                                             *
c     *      nbloks [integer]                                       *
c     *             The number of nrwblkx2*nrwblk blocks            *
c     *             in array(,,).                                   *
c     *                                                             *
c     *      rgtblk [double precision(nrwblk,nrwblk)]               *
c     *             The top right block of the ABD matrix.          *
c     *                                                             *
c     *           b [double precision((nbloks+1)*nrwblk)]           *
c     *             The right-hand side vector.                     *
c     *                                                             *
c     *       pivot [integer((nbloks+1)*nrwblk)]                    *
c     *             Work space to hold the pivoting strategy.       *
c     *                                                             *
c     *      nparts [integer]                                       *
c     *             The number of partitions to use in the          *
c     *             decomposition and solve.                        *
c     *                                                             *
c     *        work [double precision                               *
c     *                    ((nbloks+2*nparts+1)*nrwblk**2)]         *
c     *             Work space to hold fill-in and local            *
c     *             storage for BLAS.                               *
```

```
c     *                                                               *
c     *   on return                                                   *
c     *                                                               *
c     *      lftblk, array, rgtblk, work                              *
c     *             The desired decomposition of the ABD matrix.      *
c     *                                                               *
c     *             [ Note:  If iflag = -1 the factorization is       *
c     *                  not complete.                          ]     *
c     *                                                               *
c     *      nrwblk, nbloks                                           *
c     *             Unchanged.                                        *
c     *                                                               *
c     *          b [double precision((nbloks+1)*nrwblk)]              *
c     *            The solution vector (if iflag = 0).                *
c     *                                                               *
c     *       pivot [integer((nbloks+1)*nrwblk)]                      *
c     *             The pivoting strategy (if iflag = 0).             *
c     *                                                               *
c     *       iflag [integer]                                         *
c     *             =  0 on normal return                             *
c     *             = -1 if the ABD matrix is singular                *
c     *                                                               *
c     *             [ Note:  Only exact singularity is detected;      *
c     *                  iflag = 0 is not a guarantee of well-        *
c     *                  conditioning.  In the case where lftblk      *
c     *                  and rgtblk can be uncoupled, Lapack's        *
c     *                  DGBTRF/DGBCON may be used to obtain a        *
c     *                  condition estimate for the ABD matrix.       *
c     *                  Subroutines are included in ABDpack for      *
c     *                  transforming the rscale-format matrix        *
c     *                  into the correct form for input into         *
c     *                  Lapack's band routines.  See 'uncple'        *
c     *                  and 'mkband' for details.              ]     *
c     *                                                               *
c     *      nparts [integer]                                         *
c     *             Normally unchanged.  If, however, the             *
c     *             requested number of partitions would              *
c     *             result in fewer than 2 blocks of array(,,)        *
c     *             per partition (i.e. if nbloks < 2*nparts),        *
c     *             the subroutine automatically resets nparts        *
c     *             to 1 and uses non-partitioned Rescaling.          *
c     *                                                               *
c     *   SUBROUTINES CALLED:                                         *
c     *                                                               *
c     *       rscfa (lftblk, array, nrwblk, nbloks, rgtblk,           *
c     *              pivot, iflag, nparts, work)                      *
c     *                                                               *
c     *          Factors the ABD matrix using parallel Rescaling.     *
c     *          Parameters are as described above.                   *
c     *                                                               *
c     *       rscsl (lftblk, array, nrwblk, nbloks, rgtblk,           *
c     *              b, pivot, nparts, work)                          *
c     *                                                               *
c     *          Uses the factors returned by 'rscfa' to perform      *
```

```
c     *          forward elimination and back-solve on right-hand    *
c     *          side b.  Parameters are as described above.         *
c     *                                                              *
c     *   SOLVING FOR MULTIPLE RIGHT-HAND SIDES:                     *
c     *                                                              *
c     *   'rscale' is called only once for a given system A x = b.   *
c     *   If iflag = 0 the system is solved.  In order to solve for  *
c     *   a different right-hand side (i.e. A x = b'), 'rscsl' is    *
c     *   called directly.  The arrays lftblk, array, rgtblk, work,  *
c     *   and pivot contain the decomposition of A and pivoting      *
c     *   strategy on return from 'rscale' and therefore must not    *
c     *   be altered between successive calls to 'rscsl'.  b is      *
c     *   the only parameter that may be changed.                    *
c     *                                                              *
c     *   REFERENCES:                                                *
c     *                                                              *
c     *   [1]  K.R. Jackson and R.N. Pancer, The parallel solution   *
c     *        of ABD systems arising in numerical methods for       *
c     *        BVPs for ODEs, University of Toronto, Department      *
c     *        of Computer Science, Technical Report 255/91, 1992.   *
c     ***------------------------------------------------------------***
c
          call rscfa (lftblk, array, nrwblk, nbloks, rgtblk,
     *                pivot, iflag, nparts, work)
          if (iflag .eq. 0) then
             call rscsl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, pivot, nparts, work)
          end if
       return
       end
c-----------------------------------------------------------------------
       subroutine rscfa (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   pivot, iflag, nparts, work)
c
       double precision lftblk(1), array(1), rgtblk(1), work(1)
       integer nrwblk, nbloks, pivot(1), iflag, nparts
c     ***------------------------------------------------------------***
c     *   This subroutine factors the ABD matrix defined in arrays   *
c     *   lftblk, array, and rgtblk using a variant of the parallel  *
c     *   Rescaling algorithm.  On return, lftblk, array, rgtblk,    *
c     *   work, and pivot contain the decomposition of the matrix    *
c     *   and pivoting strategy used.  See comments in subroutine    *
c     *   'rscale' for further details.                              *
c     ***------------------------------------------------------------***
          integer nsquar, wk1, wk2, wk3, wk4, minblk, remblk
c
          iflag = 0
c     ***------------------------------------------------------------***
c     *      Use non-partitioned Rescaling if requested number       *
c     *      of partitions would result in fewer than 2 blocks       *
c     *      per partition.                                          *
c     ***------------------------------------------------------------***
          if (nbloks .lt. 2*nparts) then
             nparts = 1
```

```
         endif
c     ***-------------------------------------------------------***
c     *      Work-space allocation:                                *
c     *         right blocks             -   work(1)..work(wk2-1)  *
c     *         1st-level product blocks - work(wk2)..work(wk3-1)  *
c     *         2nd-level product block  - work(wk3)..work(wk4-1)  *
c     *         local storage for BLAS   - work(wk4)..end          *
c     *                                                           *
c     *      Total requirement:  nbloks*[nrwblkxnrwblk]           *
c     *                        + nparts*[nrwblkxnrwblk]           *
c     *                        +        [nrwblkxnrwblk]           *
c     *                        + nparts*[nrwblkxnrwblk]           *
c     ***-------------------------------------------------------***
         nsquar = nrwblk**2
         wk1 = 1
         wk2 = wk1 + nbloks*nsquar
         wk3 = wk2 + nparts*nsquar
         wk4 = wk3 + nsquar
c     ***-------------------------------------------------------***
c     *      Calculate minimum number of blocks per partition     *
c     *      Remaining blocks are distributed evenly among the    *
c     *      first partitions.                                    *
c     ***-------------------------------------------------------***
         minblk = nbloks/nparts
         remblk = nbloks - minblk*nparts
c     ***-------------------------------------------------------***
c     *      Three level factorization.  The factorization is     *
c     *      aborted immediatley if singularity is detected.      *
c     ***-------------------------------------------------------***
         call rscf1(array,work(wk1),work(wk2),nrwblk,pivot,iflag,
     *              minblk,remblk,nparts,work(wk4))
         if (iflag .eq. 0) then
            call rscf2(array,work(wk1),work(wk2),work(wk3),nrwblk,
     *                 pivot,iflag,minblk,remblk,nparts,work(wk4))
            if (iflag .eq. 0) then
               call rscf3(lftblk,array,work(wk1),work(wk2),work(wk3),
     *             nrwblk,nbloks,rgtblk,pivot,iflag,nparts,work(wk4))
            endif
         endif
c     ***-------------------------------------------------------***
c     *      Set iflag to -1 if exact singularity was detected.   *
c     ***-------------------------------------------------------***
         if (iflag .ne. 0) then
            iflag = -1
         endif
       return
       end
c-----------------------------------------------------------------------
       subroutine rscf1 (array, right, prodx1, nrwblk, pivot, iflag,
     *                   minblk, remblk, nparts, blaws)
c
       double precision array(nrwblk,2*nrwblk,1),
     *               right(nrwblk,nrwblk,1), prodx1(nrwblk,nrwblk,1),
     *               blaws(nrwblk,nrwblk,1)
```

```
      integer nrwblk, pivot(1), iflag, minblk, remblk, nparts
c     ***-------------------------------------------------------***
c     *   The following notation is used in the comments:         *
c     *                                                           *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)       *
c     *           R_k <=> right(1,1,k)                            *
c     *           S_k <=> prodx1(1,1,k)                           *
c     *                                                           *
c     *   In addition, the affix ' designates that the matrix is  *
c     *   transformed at the first level of the factorization.    *
c     ***-------------------------------------------------------***
          integer nsquar, kpart, kblok, base, base1, top, info
c
          nsquar = nrwblk**2
c     ***-------------------------------------------------------***
c     *      Each loop 20 iteration is independent and could      *
c     *      execute concurrently with the others.                *
c     ***-------------------------------------------------------***
C$DOACROSS SHARE (array, right, prodx1, nrwblk, pivot, iflag,
C$&               minblk, remblk, nparts, blaws, nsquar),
C$&        LOCAL (kpart, kblok, base, base1, top, info)
          do 20 kpart = 1, nparts
c     ***-------------------------------------------------------***
c     *         Rescaling starts at the second-last block-row     *
c     *         of each partition.                                *
c     ***-------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            base1 = base - 1
c     ***-------------------------------------------------------***
c     *         W_base1' <- (W_base1 - V_base1)                   *
c     ***-------------------------------------------------------***
            call DAXPY(nsquar,-1.d0,array(1,1,base1),1,
     *                              array(1,nrwblk+1,base1),1)
c     ***-------------------------------------------------------***
c     *         W_base1' <- LUfact(W_base1 - V_base1)             *
c     ***-------------------------------------------------------***
            call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,base1),nrwblk,
     *                  pivot(base1*nrwblk+1),iflag)
            if (iflag .ne. 0) return
c     ***-------------------------------------------------------***
c     *         V_base1' <- (W_base1 - V_base1)^-1 V_base1        *
c     ***-------------------------------------------------------***
            call DGETRS('N',nrwblk,nrwblk,array(1,nrwblk+1,base1),
     *                  nrwblk,pivot(base1*nrwblk+1),
     *                  array(1,1,base1),nrwblk,info)
c     ***-------------------------------------------------------***
c     *         S_kpart' <- (V_base V_base1')^T                   *
c     *                                                           *
c     *  [ Notes: 1. The transpose of the product is accumulated  *
c     *             since Lapack's DGEMM is faster multiplying in *
c     *             this mode when both matrices are dense.       *
c     *                                                           *
c     *           2. After the last call to DGEMM, the resulting  *
c     *             matrix must be transposed once.             ]  *
```

```
c     ***-------------------------------------------------------***
            call DGEMM('T','T',nrwblk,nrwblk,nrwblk,
     *                 1.d0,array(1,1,base1),nrwblk,
     *                 array(1,1,base),nrwblk,
     *                 0.d0,prodx1(1,1,kpart),nrwblk)
c     ***-------------------------------------------------------***
c     *         Each partition is now processed sequentially        *
c     *         from the third-last block row to the top.           *
c     ***-------------------------------------------------------***
            do 10 kblok = base-2, top, -1
c     ***-------------------------------------------------------***
c     *            R_kblok <- W_kblok                               *
c     *                                                             *
c     *  [ Note:  The right block must be saved in order to         *
c     *           transform subsequent right hand sides.          ]  *
c     ***-------------------------------------------------------***
            call DCOPY(nsquar,array(1,nrwblk+1,kblok),1,
     *                        right(1,1,kblok),1)
c     ***-------------------------------------------------------***
c     *         W_kblok' <- W_kblok(I + V_kblok+1') - V_kblok       *
c     ***-------------------------------------------------------***
            call DCOPY(nsquar,array(1,1,kblok+1),1,
     *                        blaws(1,1,kpart),1)
            call maddi('+',nrwblk,blaws(1,1,kpart))
            call DCOPY(nsquar,array(1,1,kblok),1,
     *                        array(1,nrwblk+1,kblok),1)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,right(1,1,kblok),nrwblk,
     *                 blaws(1,1,kpart),nrwblk,
     *                 -1.d0,array(1,nrwblk+1,kblok),nrwblk)
c     ***-------------------------------------------------------***
c     *     W_kblok' <- LUfact(W_kblok(I + V_kblok+1') - V_kblok)   *
c     ***-------------------------------------------------------***
            call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,kblok),
     *                  nrwblk,pivot(kblok*nrwblk+1),iflag)
            if (iflag .ne. 0) return
c     ***-------------------------------------------------------***
c     *         V_kblok' <- (W_kblok(I + V_kblok+1')                *
c     *                           - V_kblok)^-1 V_kblok             *
c     ***-------------------------------------------------------***
            call DGETRS('N',nrwblk,nrwblk,array(1,nrwblk+1,kblok),
     *                  nrwblk,pivot(kblok*nrwblk+1),
     *                  array(1,1,kblok),nrwblk,info)
c     ***-------------------------------------------------------***
c     *            S_kpart' <- V_kblok'^T S_kpart'                  *
c     ***-------------------------------------------------------***
            call DGEMM('T','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,array(1,1,kblok),nrwblk,
     *                 prodx1(1,1,kpart),nrwblk,
     *                 0.d0,blaws(1,1,kpart),nrwblk)
            call DCOPY(nsquar,blaws(1,1,kpart),1,
     *                        prodx1(1,1,kpart),1)
  10        continue
c     ***-------------------------------------------------------***
```

```
c     *         The final product must be transposed.                *
c     ***------------------------------------------------------------***
            call mtran(nrwblk,prodx1(1,1,kpart))
c     ***------------------------------------------------------------***
c     *         If there was an odd number of multiplications,       *
c     *         the final product also must be negated.              *
c     ***------------------------------------------------------------***
            if (mod(base-top,2) .ne. 0) then
               call mnegv(nrwblk,prodx1(1,1,kpart))
            end if
  20     continue
c     ***------------------------------------------------------------***
c     *      The second-level array right blocks are computed.       *
c     *      (This could be done concurrently.)                      *
c     ***------------------------------------------------------------***
         do 30 kpart = 1, nparts - 1
            call partx(minblk,remblk,kpart,base,top)
c     ***------------------------------------------------------------***
c     *         R_base <- W_base                                     *
c     ***------------------------------------------------------------***
            call DCOPY(nsquar,array(1,nrwblk+1,base),1,
     *                        right(1,1,base),1)
c     ***------------------------------------------------------------***
c     *         W_base' <- W_base(I + V_base+1')                     *
c     ***------------------------------------------------------------***
            call DCOPY(nsquar,array(1,1,base+1),1,
     *                        blaws(1,1,kpart),1)
            call maddi('+',nrwblk,blaws(1,1,kpart))
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,right(1,1,base),nrwblk,
     *                 blaws(1,1,kpart),nrwblk,
     *                 0.d0,array(1,nrwblk+1,base),nrwblk)
c     ***------------------------------------------------------------***
c     *         R_base-1 <- W_base(I + V_base+1')                    *
c     *                                                              *
c     *  [ Notes: 1. R_base-1 is free at this point.  It is the      *
c     *             only right block that need not be saved.         *
c     *                                                              *
c     *           2. W_base(I + V_base+1') is stored in R_base-1     *
c     *             for use during second-level processing.       ]  *
c     ***------------------------------------------------------------***
            call DCOPY(nsquar,array(1,nrwblk+1,base),1,
     *                        right(1,1,base-1),1)
  30     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine rscf2 (array, right, prodx1, prodx2, nrwblk,
     *                  pivot, iflag, minblk, remblk, nparts, blaws)
c
      double precision array(nrwblk,2*nrwblk,1),
     *               right(nrwblk,nrwblk,1), prodx1(nrwblk,nrwblk,1),
     *               prodx2(nrwblk,1), blaws(nrwblk,1)
      integer nrwblk, pivot(1), iflag, minblk, remblk, nparts
```

```
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:               *
c     *                                                                 *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)             *
c     *           R_k <=> right(1,1,k)                                  *
c     *           S_k <=> prodx1(1,1,k)                                 *
c     *           T   <=> prodx2(1,1)                                   *
c     *                                                                 *
c     *   In addition, the affix '/'' designates that the               *
c     *   matrix was/is transformed at the first/second level           *
c     *   of the factorization.                                         *
c     ***-------------------------------------------------------------***
          integer nsquar, kpart, nparts1, base, top, info
c     ***-------------------------------------------------------------***
c     *      If there is only one partition, nothing needs to be        *
c     *      done at the second level of the factorization.             *
c     ***-------------------------------------------------------------***
          if (nparts .eq. 1) return
c     ***-------------------------------------------------------------***
c     *      Rescaling starts at the second-last block-row of           *
c     *      the second-level array.                                    *
c     ***-------------------------------------------------------------***
          nsquar = nrwblk**2
          nparts1 = nparts - 1
          call partx(minblk,remblk,nparts1,base,top)
c     ***-------------------------------------------------------------***
c     *      W_base'' <- (W_base' - S_nparts-1')                        *
c     ***-------------------------------------------------------------***
          call DAXPY(nsquar,-1.d0,prodx1(1,1,nparts1),1,
     *                            array(1,nrwblk+1,base),1)
c     ***-------------------------------------------------------------***
c     *      W_base'' <- LUfact(W_base' - S_nparts-1')                  *
c     ***-------------------------------------------------------------***
          call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,base),nrwblk,
     *               pivot(base*nrwblk+1),iflag)
          if (iflag .ne. 0) return
c     ***-------------------------------------------------------------***
c     *   S_nparts-1'' <- (W_base' - S_nparts-1')^-1 S_nparts-1'        *
c     ***-------------------------------------------------------------***
          call DGETRS('N',nrwblk,nrwblk,array(1,nrwblk+1,base),
     *               nrwblk,pivot(base*nrwblk+1),
     *               prodx1(1,1,nparts1),nrwblk,info)
c     ***-------------------------------------------------------------***
c     *         T'' <- (S_nparts' S_nparts-1'')^T                       *
c     *                                                                 *
c     *  [ Notes: 1. The transpose of the product is accumulated        *
c     *             since Lapack's DGEMM is faster multiplying in       *
c     *             this mode when both matrices are dense.             *
c     *                                                                 *
c     *           2. After the last call to DGEMM, the resulting        *
c     *             matrix must be transposed once.               ]     *
c     ***-------------------------------------------------------------***
          call DGEMM('T','T',nrwblk,nrwblk,nrwblk,
     *              1.d0,prodx1(1,1,nparts1),nrwblk,
```

```
     *             prodx1(1,1,nparts),nrwblk,
     *             0.d0,prodx2,nrwblk)
c    ***-------------------------------------------------------***
c    *      The second-level array is now processed sequentially   *
c    *      from the third-last block row to the top.              *
c    ***-------------------------------------------------------***
         do 10 kpart = nparts-2, 1, -1
            call partx(minblk,remblk,kpart,base,top)
c    ***-------------------------------------------------------***
c    *         W_base'' <- W_base'(I + S_kpart+1'') - S_kpart'     *
c    *                                                             *
c    *  [ Note:  A copy of W_base' was stored in R_base-1 during   *
c    *           the first-level factorization.                ]  *
c    ***-------------------------------------------------------***
            call DCOPY(nsquar,prodx1(1,1,kpart+1),1,blaws,1)
            call maddi('+',nrwblk,blaws)
            call DCOPY(nsquar,prodx1(1,1,kpart),1,
     *                        array(1,nrwblk+1,base),1)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,right(1,1,base-1),nrwblk,blaws,nrwblk,
     *                 -1.d0,array(1,nrwblk+1,base),nrwblk)
c    ***-------------------------------------------------------***
c    *    W_base'' <- LUfact(W_base'(I + S_kpart+1'') - S_kpart')  *
c    ***-------------------------------------------------------***
            call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,base),nrwblk,
     *                  pivot(base*nrwblk+1),iflag)
            if (iflag .ne. 0) return
c    ***-------------------------------------------------------***
c    *         S_kpart'' <- (W_base'(I + S_kpart+1'')             *
c    *                               - S_kpart')^-1 S_kpart'       *
c    ***-------------------------------------------------------***
            call DGETRS('N',nrwblk,nrwblk,array(1,nrwblk+1,base),
     *                  nrwblk,pivot(base*nrwblk+1),
     *                  prodx1(1,1,kpart),nrwblk,info)
c    ***-------------------------------------------------------***
c    *         T'' <- S_kpart''^T T''                              *
c    ***-------------------------------------------------------***
            call DGEMM('T','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,prodx1(1,1,kpart),nrwblk,
     *                 prodx2,nrwblk,0.d0,blaws,nrwblk)
            call DCOPY(nsquar,blaws,1,prodx2,1)
  10     continue
c    ***-------------------------------------------------------***
c    *         The final product must be transposed.               *
c    ***-------------------------------------------------------***
         call mtran(nrwblk,prodx2)
c    ***-------------------------------------------------------***
c    *         If there was an odd number of multiplications,      *
c    *         the final product also must be negated.             *
c    ***-------------------------------------------------------***
         if (mod(nparts1,2) .ne. 0) then
            call mnegv(nrwblk,prodx2)
         end if
       return
```

```
      end
c-----------------------------------------------------------------------
      subroutine rscf3 (lftblk, array, right, prodx1, prodx2, nrwblk,
     *                  nbloks, rgtblk, pivot, iflag, nparts, blaws)
c
      double precision lftblk(nrwblk,1), array(nrwblk,2*nrwblk,1),
     *            right(nrwblk,nrwblk,1), prodx1(nrwblk,nrwblk,1),
     *            prodx2(nrwblk,1), rgtblk(nrwblk,1), blaws(nrwblk,1)
      integer nrwblk, nbloks, pivot(1), iflag, nparts
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:              *
c     *                                                                *
c     *      B_a, B_b <=> lftblk, rgtblk                               *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)            *
c     *           R_k <=> right(1,1,k)                                 *
c     *           S_k <=> prodx1(1,1,k)                                *
c     *           T   <=> prodx2(1,1)                                  *
c     *                                                                *
c     *   In addition, the affix '/'' designates that the              *
c     *   matrix was (or at least could have been) transformed         *
c     *   at the first/second level of the factorization.              *
c     *   (In order to be consistent with other variants of            *
c     *   this algorithm, all transformations to B_a and B_b           *
c     *   occur at the third level of the factorization.)              *
c     ***------------------------------------------------------------***
         integer nsquar, info
c
         nsquar = nrwblk**2
c     ***------------------------------------------------------------***
c     *      R_nbloks <- LUfact(W_nbloks)                              *
c     *                                                                *
c     *  [ Note:  The LU factorization of W_nbloks is stored           *
c     *           in R_nbloks for use below and for processing         *
c     *           subsequent right hand sides.  The pivot indices      *
c     *           are stored in pivot(1..nrwblk).                ]    *
c     ***------------------------------------------------------------***
         call DCOPY(nsquar,array(1,nrwblk+1,nbloks),1,
     *                     right(1,1,nbloks),1)
         call DGETRF(nrwblk,nrwblk,right(1,1,nbloks),nrwblk,
     *               pivot,iflag)
         if (iflag .ne. 0) return
c     ***------------------------------------------------------------***
c     *      The content of part of the third-level block-array        *
c     *      depends on whether or not paritioning was done.           *
c     ***------------------------------------------------------------***
         if (nparts .gt. 1) then
c     ***------------------------------------------------------------***
c     *      If there is more than one partition, the third-level      *
c     *      block-array is of the form  [B_a''  B_b                   *
c     *                                   T''   W_nbloks],             *
c     *      where  B_a'' = B_a' + B_a' S_1''                          *
c     *                   = B_a' (I + S_1'')                           *
c     *                   = (B_a + B_a V_1') (I + S_1'')               *
c     *                   = B_a (I + V_1') (I + S_1'')                 *
```

```
c     *                  = B_a (I + V_1' + S_1'' + V_1' S_1''),     *
c     *                                                             *
c     *      and     T'' = +/-S_nparts' S_nparts-1''... S_1''.      *
c     *                                                             *
c     *      Instead of directly factoring this 2x2 block array,    *
c     *      the reduced array [B_a'' - B_b W_nbloksˆ-1 T''] is      *
c     *      factored and its factorization is stored at the base   *
c     *      of the last partition in W_nbloks.                     *
c     *                                                             *
c     *  [ Note:  This cannot be done in the slf_lu or slf_qr       *
c     *           algorithms since W_nbloks is often numerically    *
c     *           singular at this level of the factorization.    ] *
c     ***-----------------------------------------------------------***
c
c     ***-----------------------------------------------------------***
c     *      W_nbloks <- B_a (I + V_1' + S_1'' + V_1' S_1'')        *
c     ***-----------------------------------------------------------***
            call DCOPY(nsquar,prodx1,1,blaws,1)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,array,nrwblk,prodx1,nrwblk,
     *                 1.d0,blaws,nrwblk)
            call DAXPY(nsquar,1.d0,array,1,blaws,1)
            call maddi('+',nrwblk,blaws)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,lftblk,nrwblk,blaws,nrwblk,
     *                 0.d0,array(1,nrwblk+1,nbloks),nrwblk)
c     ***-----------------------------------------------------------***
c     *         W_nbloks <- (B_a'' - B_b W_nbloksˆ-1 T'')           *
c     ***-----------------------------------------------------------***
            call DCOPY(nsquar,prodx2,1,blaws,1)
            call DGETRS('N',nrwblk,nrwblk,right(1,1,nbloks),
     *                  nrwblk,pivot,blaws,nrwblk,info)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 -1.d0,rgtblk,nrwblk,blaws,nrwblk,
     *                 1.d0,array(1,nrwblk+1,nbloks),nrwblk)
c     ***-----------------------------------------------------------***
c     *         W_nbloks <- LUfact(B_a'' - B_b W_nbloksˆ-1 T'')     *
c     ***-----------------------------------------------------------***
            call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,nbloks),nrwblk,
     *                  pivot(nbloks*nrwblk+1),iflag)
          else
c     ***-----------------------------------------------------------***
c     *      If there is only one partition, the third-level        *
c     *      block-array is of the form  [B_a'  B_b                 *
c     *                                   S_1'  W_nbloks],          *
c     *      where  B_a' = B_a + B_a V_1',                          *
c     *                                                             *
c     *      and    S_1' = +/-V_nbloks V_nbloks-1'... V_1'.         *
c     *                                                             *
c     *      Instead of directly factoring this 2x2 block array,    *
c     *      the reduced array [B_a' - B_b W_nbloksˆ-1 S_1'] is     *
c     *      factored and its factorization is stored in W_nbloks.  *
c     ***-----------------------------------------------------------***
c
```

```
c     ***-------------------------------------------------------------***
c     *         W_nbloks <- (B_a + B_a V_1')                            *
c     ***-------------------------------------------------------------***
            call DCOPY(nsquar,lftblk,1,array(1,nrwblk+1,nbloks),1)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 1.d0,lftblk,nrwblk,array,nrwblk,
     *                 1.d0,array(1,nrwblk+1,nbloks),nrwblk)
c     ***-------------------------------------------------------------***
c     *         W_nbloks <- (B_a' - B_b W_nbloks^-1 S_1')                *
c     ***-------------------------------------------------------------***
            call DCOPY(nsquar,prodx1,1,blaws,1)
            call DGETRS('N',nrwblk,nrwblk,right(1,1,nbloks),
     *                  nrwblk,pivot,blaws,nrwblk,info)
            call DGEMM('N','N',nrwblk,nrwblk,nrwblk,
     *                 -1.d0,rgtblk,nrwblk,blaws,nrwblk,
     *                 1.d0,array(1,nrwblk+1,nbloks),nrwblk)
c     ***-------------------------------------------------------------***
c     *         W_nbloks <- LUfact(B_a' - B_b W_nbloks^-1 S_1')         *
c     ***-------------------------------------------------------------***
            call DGETRF(nrwblk,nrwblk,array(1,nrwblk+1,nbloks),nrwblk,
     *                  pivot(nbloks*nrwblk+1),iflag)
          endif
       return
       end
c-----------------------------------------------------------------------
       subroutine rscsl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, pivot, nparts, work)
c
       double precision lftblk(1), array(1), rgtblk(1), b(1), work(1)
       integer nrwblk, nbloks, pivot(1), nparts
c     ***-------------------------------------------------------------***
c     *   Given the factors of ABD matrix A computed by subroutine      *
c     *   'rscfa' and stored in arrays lftblk, array, rgtblk,           *
c     *   work and pivot, this subroutine solves the linear system      *
c     *   A x = b.  b is overwritten with x.  See comments in           *
c     *   subroutine 'rscale' for further details.                      *
c     ***-------------------------------------------------------------***
          integer nsquar, wk1, wk2, wk3, wk4, minblk, remblk
c     ***-------------------------------------------------------------***
c     *      Work-space allocation:                                     *
c     *         right blocks             -   work(1)..work(wk2-1)     *
c     *         1st-level product blocks - work(wk2)..work(wk3-1)     *
c     *         2nd-level product block  - work(wk3)..work(wk4-1)     *
c     *         local storage for BLAS   - work(wk4)..end             *
c     *                                                               *
c     *      Total requirement:  nbloks*[nrwblkxnrwblk]               *
c     *                        + nparts*[nrwblkxnrwblk]               *
c     *                        +        [nrwblkxnrwblk]               *
c     *                        + nparts*[nrwblkxnrwblk]               *
c     ***-------------------------------------------------------------***
          nsquar = nrwblk**2
          wk1 = 1
          wk2 = wk1 + nbloks*nsquar
          wk3 = wk2 + nparts*nsquar
```

```
         wk4 = wk3 + nsquar
c    ***---------------------------------------------------------***
c    *      Calculate minimum number of blocks per partition       *
c    *      Remaining blocks are distributed evenly among the      *
c    *      first partitions.                                      *
c    ***---------------------------------------------------------***
         minblk = nbloks/nparts
         remblk = nbloks - minblk*nparts
c    ***---------------------------------------------------------***
c    *      Three level forward elimination.                       *
c    ***---------------------------------------------------------***
         call rscsf1(array,work(wk1),nrwblk,b,
     *               pivot,minblk,remblk,nparts,work(wk4))
         call rscsf2(array,work(wk1),work(wk2),nrwblk,b,
     *               pivot,minblk,remblk,nparts,work(wk4))
         call rscsf3(lftblk,array,work(wk1),nrwblk,rgtblk,
     *               b,b,pivot,minblk,remblk,nparts,work(wk4))
c    ***---------------------------------------------------------***
c    *      Three level back-solve.                                *
c    ***---------------------------------------------------------***
         call rscsb3(array,work(wk1),work(wk2),work(wk3),
     *               nrwblk,nbloks,b,b,pivot,nparts,work(wk4))
         call rscsb2(array,work(wk2),nrwblk,b,
     *               minblk,remblk,nparts,work(wk4))
         call rscsb1(array,nrwblk,b,
     *               minblk,remblk,nparts,work(wk4))
      return
      end
c-----------------------------------------------------------------------
      subroutine rscsf1 (array, right, nrwblk, phi,
     *                   pivot, minblk, remblk, nparts, blaws)
c
      double precision array(nrwblk,2*nrwblk,1),
     *         right(nrwblk,nrwblk,1), phi(1), blaws(nrwblk,nrwblk,1)
      integer nrwblk, pivot(1), minblk, remblk, nparts
c    ***---------------------------------------------------------***
c    *   The following notation is used in the comments:           *
c    *                                                             *
c    *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)         *
c    *           R_k <=> right(1,1,k)                              *
c    *         phi_k <=> phi(k*nrwblk+1)                           *
c    *                                                             *
c    *   In addition, the affix ' designates that the vector is    *
c    *   transformed at the first level of the forward solve.      *
c    ***---------------------------------------------------------***
         integer kpart, kblok, base, base1, top, info
c    ***---------------------------------------------------------***
c    *      Each loop 30 iteration is independent and could        *
c    *      execute concurrently with the others.                  *
c    ***---------------------------------------------------------***
C$DOACROSS SHARE (array, right, nrwblk, phi,
C$&               pivot, minblk, remblk, nparts, blaws),
C$&        LOCAL (kpart, kblok, base, base1, top, info)
         do 30 kpart = 1, nparts
```

```
c     ***-------------------------------------------------------------***
c     *         Forward elimination starts at the second-last         *
c     *         block-row of each partition.                          *
c     ***-------------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            base1 = base - 1
c     ***-------------------------------------------------------------***
c     *         phi_base1' <- (W_base1 - V_base1)^-1 phi_base1        *
c     ***-------------------------------------------------------------***
            call DGETRS('N',nrwblk,1,array(1,nrwblk+1,base1),
     *                  nrwblk,pivot(base1*nrwblk+1),
     *                  phi(base1*nrwblk+1),nrwblk,info)
c     ***-------------------------------------------------------------***
c     *         Each partition is now processed sequentially          *
c     *         from the third-last block row to the top.             *
c     ***-------------------------------------------------------------***
            do 10 kblok = base-2, top, -1
c     ***-------------------------------------------------------------***
c     *        phi_kblok' <- phi_kblok + R_kblok phi_kblok+1'         *
c     ***-------------------------------------------------------------***
               call DGEMV('N',nrwblk,nrwblk,
     *                    1.d0,right(1,1,kblok),nrwblk,
     *                    phi((kblok+1)*nrwblk+1),1,
     *                    1.d0,phi(kblok*nrwblk+1),1)
c     ***-------------------------------------------------------------***
c     *        phi_kblok' <- (W_kblok(I + V_kblok+1')                 *
c     *                               - V_kblok)^-1 phi_kblok'        *
c     ***-------------------------------------------------------------***
               call DGETRS('N',nrwblk,1,array(1,nrwblk+1,kblok),
     *                     nrwblk,pivot(kblok*nrwblk+1),
     *                     phi(kblok*nrwblk+1),nrwblk,info)
  10        continue
c     ***-------------------------------------------------------------***
c     *      phi_base is now computed.  First, phi_top is stored      *
c     *      in a temporary vector (temp <- phi_top).                 *
c     ***-------------------------------------------------------------***
            call DCOPY(nrwblk,phi(top*nrwblk+1),1,blaws(1,1,kpart),1)
c     ***-------------------------------------------------------------***
c     *      The temporary vector is then processed sequentially      *
c     *      from the second block-row to the bottom.                 *
c     ***-------------------------------------------------------------***
            do 20 kblok = top+1, base-1
c     ***-------------------------------------------------------------***
c     *            temp <- phi_kblok' - V_kblok' temp                 *
c     ***-------------------------------------------------------------***
               call DCOPY(nrwblk,phi(kblok*nrwblk+1),1,
     *                           blaws(1,2,kpart),1)
               call DGEMV('N',nrwblk,nrwblk,
     *                    -1.d0,array(1,1,kblok),nrwblk,
     *                    blaws(1,1,kpart),1,1.d0,blaws(1,2,kpart),1)
               call DCOPY(nrwblk,blaws(1,2,kpart),1,
     *                           blaws(1,1,kpart),1)
  20        continue
c     ***-------------------------------------------------------------***
```

```
c     *         phi_base' <- phi_base - V_base temp                  *
c     ***-------------------------------------------------------------***
            call DGEMV('N',nrwblk,nrwblk,
     *                 -1.d0,array(1,1,base),nrwblk,
     *                 blaws(1,1,kpart),1,1.d0,phi(base*nrwblk+1),1)
  30     continue
c     ***-------------------------------------------------------------***
c     *      Finally, phi_base is updated in each partition by       *
c     *      computing a vector sum across partition boundaries.     *
c     *      (This could be done concurrently.)                      *
c     ***-------------------------------------------------------------***
         do 40 kpart = 1, nparts - 1
c     ***-------------------------------------------------------------***
c     *         phi_base' <- phi_base' + R_base phi_base+1'           *
c     ***-------------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            call DGEMV('N',nrwblk,nrwblk,
     *                 1.d0,right(1,1,base),nrwblk,
     *                 phi((base+1)*nrwblk+1),1,
     *                 1.d0,phi(base*nrwblk+1),1)
  40     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine rscsf2 (array, right, prodx1, nrwblk, phi,
     *                   pivot, minblk, remblk, nparts, blaws)
c
      double precision array(nrwblk,2*nrwblk,1),
     *                 right(nrwblk,nrwblk,1), phi(1),
     *                 prodx1(nrwblk,nrwblk,1), blaws(nrwblk,1)
      integer nrwblk, pivot(1), minblk, remblk, nparts
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)           *
c     *           R_k <=> right(1,1,k)                                *
c     *           S_k <=> prodx1(1,1,k)                               *
c     *         phi_k <=> phi(k*nrwblk+1)                             *
c     *                                                               *
c     *   In addition, the affix '/'' designates that the             *
c     *   vector was/is transformed at the first/second level         *
c     *   of the forward solve.                                       *
c     ***-------------------------------------------------------------***
         integer kpart, base, basep, top, info
c     ***-------------------------------------------------------------***
c     *      If there is only one partition, nothing needs to be      *
c     *      done at the second level of the forward solve.           *
c     ***-------------------------------------------------------------***
         if (nparts .eq. 1) return
c     ***-------------------------------------------------------------***
c     *      Forward elimination starts at the second-last            *
c     *      block-row of the second-level array.                     *
c     ***-------------------------------------------------------------***
         call partx(minblk,remblk,nparts-1,base,top)
```

```
c     ***-------------------------------------------------------***
c     *      phi_base'' <- (W_base' - S_nparts-1')^-1 phi_base'      *
c     ***-------------------------------------------------------***
          call DGETRS('N',nrwblk,1,array(1,nrwblk+1,base),
     *               nrwblk,pivot(base*nrwblk+1),
     *               phi(base*nrwblk+1),nrwblk,info)
c     ***-------------------------------------------------------***
c     *      Forward elimination now proceeds sequentially          *
c     *      from the third-last block row to the top of the        *
c     *      second-level array.                                    *
c     ***-------------------------------------------------------***
          do 10 kpart = nparts-2, 1, -1
             basep = base
             call partx(minblk,remblk,kpart,base,top)
c     ***-------------------------------------------------------***
c     *       phi_base'' <- phi_base' + W_base' phi_basep''         *
c     *                                                             *
c     *  [ Note:  A copy of W_base' was stored in R_base-1 during   *
c     *           the first-level factorization.                ]  *
c     ***-------------------------------------------------------***
             call DGEMV('N',nrwblk,nrwblk,
     *               1.d0,right(1,1,base-1),nrwblk,
     *               phi(basep*nrwblk+1),1,1.d0,phi(base*nrwblk+1),1)
c     ***-------------------------------------------------------***
c     *      phi_base'' <- (W_base'(I + S_kpart+1'')                *
c     *                              - S_kpart')^-1 phi_base''      *
c     ***-------------------------------------------------------***
             call DGETRS('N',nrwblk,1,array(1,nrwblk+1,base),
     *                  nrwblk,pivot(base*nrwblk+1),
     *                  phi(base*nrwblk+1),nrwblk,info)
  10      continue
c     ***-------------------------------------------------------***
c     *   phi_base_nparts'' is now computed.  First, phi_base_1''   *
c     *   is stored in a temporary vector (temp <- phi_base_1'').   *
c     ***-------------------------------------------------------***
          call partx(minblk,remblk,1,base,top)
          call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
c     ***-------------------------------------------------------***
c     *      The temporary vector is then processed sequentially    *
c     *      from the second block-row to the bottom of the         *
c     *      second-level array.                                    *
c     ***-------------------------------------------------------***
          do 20 kpart = 2, nparts-1
             call partx(minblk,remblk,kpart,base,top)
c     ***-------------------------------------------------------***
c     *         temp <- phi_base'' - S_kpart'' temp                 *
c     ***-------------------------------------------------------***
             call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws(1,2),1)
             call DGEMV('N',nrwblk,nrwblk,
     *                 -1.d0,prodx1(1,1,kpart),nrwblk,
     *                 blaws,1,1.d0,blaws(1,2),1)
             call DCOPY(nrwblk,blaws(1,2),1,blaws,1)
  20      continue
c     ***-------------------------------------------------------***
```

```
c     *   phi_base_nparts'' <- phi_base_nparts' - S_nparts' temp     *
c     ***------------------------------------------------------------***
          call partx(minblk,remblk,nparts,base,top)
          call DGEMV('N',nrwblk,nrwblk,
     *              -1.d0,prodx1(1,1,nparts),nrwblk,
     *              blaws,1,1.d0,phi(base*nrwblk+1),1)
       return
       end
c-----------------------------------------------------------------------
       subroutine rscsf3 (lftblk, array, right, nrwblk, rgtblk,
     *               beta, phi, pivot, minblk, remblk, nparts, blaws)
c
       double precision lftblk(nrwblk,1), array(nrwblk,2*nrwblk,1),
     *                 right(nrwblk,nrwblk,1), rgtblk(nrwblk,1),
     *                 beta(1), phi(1), blaws(nrwblk,1)
       integer nrwblk, pivot(1), minblk, remblk, nparts
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:              *
c     *                                                                *
c     *      B_a, B_b <=> lftblk, rgtblk                               *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)            *
c     *           R_k <=> right(1,1,k)                                 *
c     *          beta <=> beta(1) (<=> phi_0)                          *
c     *         phi_k <=> phi(k*nrwblk+1), k >= 1                      *
c     *                                                                *
c     *   In addition, the affix '/'' designates that the              *
c     *   vector was (or at least could have been) transformed         *
c     *   at the first/second level of the forward solve.              *
c     *   (In order to be consistent with other variants of            *
c     *   this algorithm, all transformations to beta occur            *
c     *   at the third level of the forward solve.)                    *
c     ***------------------------------------------------------------***
          integer base, top, info
c
          if (nparts .gt. 1) then
c     ***------------------------------------------------------------***
c     *      If there is more than one partition, the third-level      *
c     *      system is of the form                                     *
c     *                                                                *
c     *       [B_a''  B_b        [y_a  = [beta''                      *
c     *         T''   W_nbloks ]  y_b]    phi_base_nparts''],         *
c     *                                                                *
c     *      where B_a'', T'', and phi_base_nparts'' are as            *
c     *      described in rscf3.f and rscsf2.f, and                    *
c     *                                                                *
c     *         beta'' = beta' + B_a' phi_base_1''                     *
c     *                = beta' + (B_a + B_a V_1') phi_base_1''         *
c     *                = beta + B_a phi_1'                             *
c     *                       + (B_a + B_a V_1') phi_base_1''          *
c     *                = beta + B_a (phi_1' + phi_base_1''             *
c     *                                    + V_1' phi_base_1'')        *
c     *                                                                *
c     *      The right-hand side of the reduced nrwblkxnrwblk          *
c     *      system for y_a (see rscf3.f) is then                      *
```

```
c     *                                                               *
c     *      beta''' = beta'' - B_b W_nbloks^-1 phi_base_nparts''.    *
c     ***-----------------------------------------------------------***
            call partx(minblk,remblk,1,base,top)
c     ***-----------------------------------------------------------***
c     *      beta'' <- beta + B_a (phi_1' + phi_base_1''              *
c     *                                   + V_1' phi_base_1'')        *
c     ***-----------------------------------------------------------***
            call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
            call DGEMV('N',nrwblk,nrwblk,1.d0,array,nrwblk,
     *                 phi(base*nrwblk+1),1,1.d0,blaws,1)
            call DAXPY(nrwblk,1.d0,phi(nrwblk+1),1,blaws,1)
            call DGEMV('N',nrwblk,nrwblk,1.d0,lftblk,nrwblk,
     *                 blaws,1,1.d0,beta,1)
c     ***-----------------------------------------------------------***
c     *      beta''' <- beta'' - B_b W_nbloks^-1 phi_base_nparts''    *
c     *                                                               *
c     *  [ Note:  The LU factorization and pivot indices              *
c     *           for W_nbloks were stored in R_nbloks and            *
c     *           pivot(1..nrwblk), respectively, during the          *
c     *           third level of the factorization.               ]   *
c     ***-----------------------------------------------------------***
            call partx(minblk,remblk,nparts,base,top)
            call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
            call DGETRS('N',nrwblk,1,right(1,1,base),nrwblk,
     *                  pivot,blaws,nrwblk,info)
            call DGEMV('N',nrwblk,nrwblk,-1.d0,rgtblk,nrwblk,
     *                 blaws,1,1.d0,beta,1)
         else
c     ***-----------------------------------------------------------***
c     *      If there is only one partition, the third-level          *
c     *      system is of the form                                    *
c     *                                                               *
c     *             [B_a'  B_b        [y_a  = [beta'                 *
c     *              S_1'  W_nbloks ]  y_b]    phi_base_1'],         *
c     *                                                               *
c     *      where B_a', S_1' are as described in rscf3.f, and        *
c     *                                                               *
c     *         beta' = beta + B_a phi_1'                             *
c     *                                                               *
c     *      The right-hand side of the reduced nrwblkxnrwblk         *
c     *      system for y_a (see rscf3.f) is then                     *
c     *                                                               *
c     *         beta''' = beta' - B_b W_nbloks^-1 phi_base_1'.         *
c     ***-----------------------------------------------------------***
            call partx(minblk,remblk,1,base,top)
c     ***-----------------------------------------------------------***
c     *         beta' <- beta + B_a phi_1'                            *
c     ***-----------------------------------------------------------***
            call DGEMV('N',nrwblk,nrwblk,1.d0,lftblk,nrwblk,
     *                 phi(nrwblk+1),1,1.d0,beta,1)
c     ***-----------------------------------------------------------***
c     *         beta''' <- beta' - B_b W_nbloks^-1 phi_base_1'         *
c     ***-----------------------------------------------------------***
```

```
            call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
            call DGETRS('N',nrwblk,1,right(1,1,base),nrwblk,
     *                  pivot,blaws,nrwblk,info)
            call DGEMV('N',nrwblk,nrwblk,-1.d0,rgtblk,nrwblk,
     *                 blaws,1,1.d0,beta,1)
         endif
       return
       end
c-----------------------------------------------------------------------
       subroutine rscsb3 (array, right, prodx1, prodx2, nrwblk,
     *                    nbloks, beta, phi, pivot, nparts, blaws)
c
       double precision array(nrwblk,2*nrwblk,1),
     *              right(nrwblk,nrwblk,1), prodx1(nrwblk,nrwblk,1),
     *              prodx2(nrwblk,1), beta(1), phi(1), blaws(nrwblk,1)
       integer nrwblk, nbloks, pivot(1), nparts
c      ***------------------------------------------------------------***
c      *   The following notation is used in the comments:              *
c      *                                                                *
c      *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)            *
c      *           R_k <=> right(1,1,k)                                 *
c      *           S_k <=> prodx1(1,1,k)                                *
c      *           T   <=> prodx2(1,1)                                  *
c      *          beta <=> beta(1) (<=> phi_0)                          *
c      *         phi_k <=> phi(k*nrwblk+1), k >= 1                      *
c      *                                                                *
c      *   Since [beta; phi] is overwritten with the solution,          *
c      *                                                                *
c      *      y_a <=> y_0      <=> beta                                 *
c      *              y_k      <=> phi_k,  k = 1, nbloks-1              *
c      *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)                 *
c      *                                                                *
c      *   In addition, the affix ''' designates that the solution      *
c      *   vector is obtained at the third level of the back-solve.     *
c      ***------------------------------------------------------------***
           integer info
c      ***------------------------------------------------------------***
c      *   First y_a''' is obtained by solving the reduced system       *
c      *                                                                *
c      *      [Ba'' - Bb W_nbloksˆ-1 T''] y_a''' = beta'''  or          *
c      *      [Ba' - Bb W_nbloksˆ-1 S_1'] y_a''' = beta'''              *
c      *                                                                *
c      *   if nparts > 1, or nparts = 1, respectively.                  *
c      *                                                                *
c      *   In either case, the factorization and pivot indices          *
c      *   for the reduced system are stored in W_nbloks and            *
c      *   pivot_nbloks, respectively.                                  *
c      *                                                                *
c      *   Note that y_a''' _is not_ y_a.  Rescaling results in         *
c      *   a change of variable which is undone at levels 2 and 1       *
c      *   of the back-solve.                                           *
c      ***------------------------------------------------------------***
           call DGETRS('N',nrwblk,1,array(1,nrwblk+1,nbloks),nrwblk,
     *                 pivot(nbloks*nrwblk+1),beta(1),nrwblk,info)
```

```
c     ***-------------------------------------------------------------***
c     *   Next, y_b''' is obtained from                                 *
c     *                                                                 *
c     *       W_nbloks y_b''' = phi_nbloks'' - T'' y_a'''  or           *
c     *       W_nbloks y_b''' = phi_nbloks' - S_1' y_a'''               *
c     *                                                                 *
c     *   if nparts > 1, or nparts = 1, respectively.                   *
c     *                                                                 *
c     *   Note that y_b''' _is_ y_b.  Rescaling at levels 1 and 2       *
c     *   of the factorization did not change y_b.                      *
c     ***-------------------------------------------------------------***
          if (nparts .gt. 1) then
             call DGEMV('N',nrwblk,nrwblk,-1.d0,prodx2,nrwblk,
     *                  beta,1,1.d0,phi(nbloks*nrwblk+1),1)
          else
             call DGEMV('N',nrwblk,nrwblk,-1.d0,prodx1,nrwblk,
     *                  beta,1,1.d0,phi(nbloks*nrwblk+1),1)
          endif
          call DGETRS('N',nrwblk,1,right(1,1,nbloks),nrwblk,
     *                pivot,phi(nbloks*nrwblk+1),nrwblk,info)
       return
       end
c-----------------------------------------------------------------------
       subroutine rscsb2 (array, prodx1, nrwblk, phi,
     *                    minblk, remblk, nparts, blaws)
c
       double precision array(nrwblk,2*nrwblk,1), phi(1),
     *                  prodx1(nrwblk,nrwblk,1), blaws(nrwblk,1)
       integer nrwblk, minblk, remblk, nparts
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:               *
c     *                                                                 *
c     *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)             *
c     *           S_k <=> prodx1(1,1,k)                                 *
c     *         phi_k <=> phi(k*nrwblk+1)                               *
c     *                                                                 *
c     *   Since phi is overwritten with the solution,                   *
c     *                                                                 *
c     *      y_a <=> y_0      <=> phi_0  (<=> beta in rscsb3.f)         *
c     *              y_k      <=> phi_k,  k = 1, nbloks-1               *
c     *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)                  *
c     *                                                                 *
c     *   In addition, the affix '''/'' designates that the             *
c     *   solution vector was/is obtained at the third/second           *
c     *   level of the back-solve.                                      *
c     ***-------------------------------------------------------------***
          integer kpart, base, basep, top
c     ***-------------------------------------------------------------***
c     *      If there is only one partition, nothing needs to be        *
c     *      done at the second level of the back-solve.                *
c     ***-------------------------------------------------------------***
          if (nparts .eq. 1) return
c     ***-------------------------------------------------------------***
c     *      Back-solve starts at the first block-row of the            *
```

```
c    *      second-level system and proceeds downward sequentially  *
c    *      to the second-last block-row.                           *
c    ***------------------------------------------------------------***
         base = 0
         do 10 kpart = 1, nparts-1
            basep = base
            call partx(minblk,remblk,kpart,base,top)
c    ***------------------------------------------------------------***
c    *         y_base'' <- phi_base - S_kpart y_basep''             *
c    ***------------------------------------------------------------***
            call DGEMV('N',nrwblk,nrwblk,
     *                 -1.d0,prodx1(1,1,kpart),nrwblk,
     *                 phi(basep*nrwblk+1),1,1.d0,
     *                 phi(base*nrwblk+1),1)
c    ***------------------------------------------------------------***
c    *      The change of variable due to second-level rescaling    *
c    *      is now undone:  y_basep'' <- y_basep'' - y_base''       *
c    ***------------------------------------------------------------***
            call DAXPY(nrwblk,-1.d0,phi(base*nrwblk+1),1,
     *                              phi(basep*nrwblk+1),1)
  10     continue
       return
       end
c---------------------------------------------------------------------
       subroutine rscsb1 (array, nrwblk, phi,
     *                   minblk, remblk, nparts, blaws)
c
       double precision array(nrwblk,2*nrwblk,1), phi(1),
     *                  blaws(nrwblk,nrwblk,1)
       integer nrwblk, minblk, remblk, nparts
c    ***------------------------------------------------------------***
c    *   The following notation is used in the comments:            *
c    *                                                              *
c    *      V_k, W_k <=> array(1,1,k), array(1,nrwblk+1,k)          *
c    *         phi_k <=> phi(k*nrwblk+1)                            *
c    *                                                              *
c    *   Since phi is overwritten with the solution,                *
c    *                                                              *
c    *      y_a <=> y_0      <=> phi_0  (<=> beta in rscsb3.f)      *
c    *              y_k      <=> phi_k,  k = 1, nbloks-1            *
c    *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)               *
c    *                                                              *
c    *   In addition, the affix ''/'' designates that the           *
c    *   solution vector was/is obtained at the second/first        *
c    *   level of the back-solve.                                   *
c    ***------------------------------------------------------------***
         integer kpart, kblok, base, top
c    ***------------------------------------------------------------***
c    *      Each loop 20 iteration is independent and could         *
c    *      execute concurrently with the others.                   *
c    ***------------------------------------------------------------***
C$DOACROSS SHARE (array, nrwblk, phi,
C$&               minblk, remblk, nparts, blaws),
C$&        LOCAL (kpart, kblok, base, top)
```

```
         do 20 kpart = 1, nparts
c    ***-------------------------------------------------------------***
c    *      Back-solve starts at the first block-row of each         *
c    *      partition and proceeds downward sequentially to          *
c    *      the second-last block-row.                               *
c    ***-------------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            do 10 kblok = top, base-1
c    ***-------------------------------------------------------------***
c    *            y_kblok' = phi_kblok - V_kblok y_kblok-1'           *
c    ***-------------------------------------------------------------***
               call DGEMV('N',nrwblk,nrwblk,
     *                    -1.d0,array(1,1,kblok),nrwblk,
     *                    phi((kblok-1)*nrwblk+1),1,1.d0,
     *                    phi(kblok*nrwblk+1),1)
c    ***-------------------------------------------------------------***
c    *      The change of variable due to first-level rescaling      *
c    *      is now undone:  y_kblok-1' <- y_kblok-1' - y_kblok'      *
c    ***-------------------------------------------------------------***
               call DAXPY(nrwblk,-1.d0,phi(kblok*nrwblk+1),1,
     *                                 phi((kblok-1)*nrwblk+1),1)
  10        continue
  20     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine maddi (sgn, n, A)
c
      character*1 sgn
      double precision A(1)
      integer n
c    ***-------------------------------------------------------------***
c    *   'maddi' overwrites A with (I + A) if sgn .eq. '+' or        *
c    *   (I - A) if sgn .eq. '-'.                                    *
c    *                                                               *
c    *    on entry                                                   *
c    *                                                               *
c    *         sgn [character*1]                                     *
c    *             '+' or '-' as above.                              *
c    *                                                               *
c    *           n [integer]                                         *
c    *             The (implicit) number of rows and columns         *
c    *             in A.  A is accessed as a 1D array inside         *
c    *             this subroutine.                                  *
c    *                                                               *
c    *           A [double precision(n**2)]                          *
c    *             The matrix to be transformed.                     *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *         sgn [character*1]                                     *
c    *             Unchanged.                                        *
c    *                                                               *
c    *           n [integer]                                         *
```

```
c    *             Unchanged.                                        *
c    *                                                               *
c    *           A [double precision(n**2)]                          *
c    *             Overwritten with either (I + A) or (I - A).       *
c    ***---------------------------------------------------------------***
         logical plus, minus, LSAME
         integer i, k, nsquar
c
         nsquar = n**2
         plus = LSAME (sgn, '+')
         minus = LSAME (sgn, '-')
         if ( plus ) then
            do 10 k = 1, n
               i = (k-1)*n + k
               A(i) = A(i) + 1.d0
  10        continue
         else if ( minus ) then
            call mnegv (n, A)
            do 20 k = 1, n
               i = (k-1)*n + k
               A(i) = A(i) + 1.d0
  20        continue
         else
            write(6,*) ' *** maddi: sgn not understood '
         end if
      return
      end
c-----------------------------------------------------------------------
      subroutine mnegv (n, A)
c
      double precision A(1)
      integer n
c    ***---------------------------------------------------------------***
c    *   'mnegv' overwrites A with -A.                               *
c    *                                                               *
c    *    on entry                                                   *
c    *                                                               *
c    *           n [integer]                                         *
c    *             The (implicit) number of rows and columns         *
c    *             in A.  A is accessed as a 1D array inside         *
c    *             this subroutine.                                  *
c    *                                                               *
c    *           A [double precision(n**2)]                          *
c    *             The matrix to be negated.                         *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *           n [integer]                                         *
c    *             Unchanged.                                        *
c    *                                                               *
c    *           A [double precision(n**2)]                          *
c    *             Overwritten with -A.                              *
c    ***---------------------------------------------------------------***
         integer k, nsquar
```

```
c
         nsquar = n**2
         do 10 k = 1, nsquar
            A(k) = - A(k)
  10     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine mtran (n, A)
c
      double precision A(n,1)
      integer n
c     ***---------------------------------------------------------------***
c     *   'mtran' overwrites A with transpose(A).                         *
c     *                                                                   *
c     *    on entry                                                       *
c     *                                                                   *
c     *           n [integer]                                             *
c     *             The number of rows and columns in A.                  *
c     *             (If A is dimensioned as a 2D array in                 *
c     *             the calling (sub) program, n must be the              *
c     *             leading dimension.)                                   *
c     *                                                                   *
c     *           A [double precision(n,n)]                               *
c     *             The matrix to be transposed.                          *
c     *                                                                   *
c     *   on return                                                       *
c     *                                                                   *
c     *           n [integer]                                             *
c     *             Unchanged.                                            *
c     *                                                                   *
c     *           A [double precision(n,n)]                               *
c     *             The transpose of A.                                   *
c     ***---------------------------------------------------------------***
         double precision temp
         integer i, j
c
         do 20 j = 1, n
            do 10 i = j+1, n
               temp = A(i,j)
               A(i,j) = A(j,i)
               A(j,i) = temp
  10        continue
  20     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine partx (minblk, remblk, k, base, top)
c
      integer minblk, remblk, k, base, top
c     ***---------------------------------------------------------------***
c     *   'partx' calculates the index of the base and top block          *
c     *   of the k-th partition.  The indexing scheme assumes that        *
c     *   nbloks >= 2*nparts, so minblk >= 2 and remblk >= 0.             *
```

```
c    *                                                               *
c    *    on entry                                                   *
c    *                                                               *
c    *      minblk [integer]                                         *
c    *             minimum number of blocks/partition                *
c    *             (minblk = nbloks/nparts)                          *
c    *                                                               *
c    *      remblk [integer]                                         *
c    *             first remblk partitions have minblk+1 blocks      *
c    *             (remblk = nbloks - minblk*nparts)                 *
c    *                                                               *
c    *           k [integer]                                         *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *        base [integer]                                         *
c    *             index of base block of k-th partition             *
c    *                                                               *
c    *         top [integer]                                         *
c    *             index of top block of k-th partition              *
c    ***-------------------------------------------------------------***
         integer khigh, klow
c
         if (k .le. remblk) then
            khigh = k
            klow = 0
         else
            khigh = remblk
            klow = k - khigh
         endif
         base = khigh*(minblk+1) + klow*minblk
         top = base - minblk + 1
         if (k .le. remblk) then
            top = top - 1
         endif
       return
       end
```

E.2 *SLF*-LU

```
      subroutine slf_lu (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, pivot, iflag, nparts, work)
c
      double precision lftblk(1), array(1), rgtblk(1), b(1), work(1)
      integer nrwblk, nbloks, pivot(1), iflag, nparts
c     ***------------------------------------------------------------***
c     *   This subroutine solves the linear system A x = b where     *
c     *   A is an Almost Block Diagonal matrix of the form           *
c     *                                                              *
c     *             lftblk                    rgtblk                 *
c     *                    a                                         *
c     *             bgnblk r                                         *
c     *                    r                                         *
c     *                    a   a                                     *
c     *                    y   r                                     *
c     *                  (,,1) r                                     *
c     *                        a   .                                 *
c     *                        y     .                               *
c     *                      (,,2)     .                             *
c     *                                  a                           *
c     *                                  r                           *
c     *                                  r                           *
c     *                                  a                           *
c     *                                  y   endblk                  *
c     *                           (,,nbloks-1)                       *
c     *                                                              *
c     *   lftblk and rgtblk are each nrwblkxnrwblk, array(,,k)       *
c     *   is 2*nrwblkxnrwblk, k = 1..nbloks, endblk/bgnblk alias     *
c     *   the upper/lower nrwblkxnrwblk block of array(,,nbloks),    *
c     *   bgnblk overlaps the first nrwblk rows of array(,,1),       *
c     *   {[array(,,k) array(,,k+1)], k = 1..nbloks-2} overlap       *
c     *   by nrwblk rows each, and endblk overlaps the last nrwblk   *
c     *   rows of array(,,nbloks-1).  The linear system is square    *
c     *   and of order (nbloks+1)*nrwblk.                            *
c     *                                                              *
c     *  [ Note:  ABDs often arise in other forms.  For example,     *
c     *       lftblk and rgtblk may be uncoupled so that lftblk      *
c     *       appears at the top of the matrix and rgtblk appears    *
c     *       at the bottom.  Also, the blocks in array(,,) are      *
c     *       often arranged so that array(,,k) holds the left       *
c     *       and right blocks in block-row k.  In these cases,      *
c     *       the ABD system first can be transformed into the       *
c     *       correct form for input to 'slf_lu' using auxiliary     *
c     *       routines included with this package.  See 'couple'     *
c     *       and 'rotcw' for details.  Alternatively, 'slfluc',     *
c     *       a modified version of 'slf_lu' that incorporates       *
c     *       both 'couple' and 'rotcw', can be used.           ]    *
c     *                                                              *
c     *   THE ALGORITHM:                                             *
c     *                                                              *
c     *   The system is decomposed and solved using a variant        *
c     *   of the parallel SLF-LU algorithm described in [1].         *
```

```
c     *   Parallelism is achieved by slicing the system into         *
c     *   'nparts' partitions in such a way that each partition     *
c     *   can be processed independently.  Assuming at least one    *
c     *   processor is available per partition, a speed-up of S     *
c     *   (over sequential SLF-LU) may be attained where            *
c     *                                                             *
c     *            1 <= S < nparts,                                 *
c     *                                                             *
c     *       with  S = 1       if  nbloks < 2*nparts,              *
c     *        and  S ~ nparts  if  nparts << nbloks/nparts.        *
c     *                                                             *
c     *   In other words, for systems of sufficiently high order,   *
c     *   speed-up is approximately linear with respect to nparts   *
c     *   when nparts is sufficiently small.  Sample problems and   *
c     *   timing benchmarks are included with this package.         *
c     *                                                             *
c     *   PARAMETERS:                                               *
c     *                                                             *
c     *    on entry                                                 *
c     *                                                             *
c     *      lftblk [double precision(nrwblk,nrwblk)]               *
c     *             The top left block of the ABD matrix.           *
c     *                                                             *
c     *       array [double precision(2*nrwblk,nrwblk,nbloks)]      *
c     *             array(,,k), k = 1..nbloks-1, contains the       *
c     *             k-th 2*nrwblkxnrwblk block of the ABD matrix    *
c     *             as described above.  array(,,nbloks) contains   *
c     *             endblk/bgnblk as described above.               *
c     *                                                             *
c     *      nrwblk [integer]                                       *
c     *             The number of columns in lftblk, array(,,k)     *
c     *             and rgtblk.  The number of rows in              *
c     *             lftblk and rgtblk.  There are 2*nrwblk          *
c     *             rows in array(,,k).                             *
c     *                                                             *
c     *      nbloks [integer]                                       *
c     *             The number of 2*nrwblkxnrwblk blocks            *
c     *             in array(,,).                                   *
c     *                                                             *
c     *      rgtblk [double precision(nrwblk,nrwblk)]               *
c     *             The top right block of the ABD matrix.          *
c     *                                                             *
c     *           b [double precision((nbloks+1)*nrwblk)]           *
c     *             The right-hand side vector.                     *
c     *                                                             *
c     *       pivot [integer((nbloks+2)*nrwblk)]                    *
c     *             Work space to hold the pivoting strategy.       *
c     *                                                             *
c     *      nparts [integer]                                       *
c     *             The number of partitions to use in the          *
c     *             decomposition and solve.                        *
c     *                                                             *
c     *        work [double precision(2*nparts*nrwblk +             *
c     *                      (2*nbloks+4*nparts+4)*nrwblk**2)]      *
```

```
c     *             Work space to hold fill-in and local             *
c     *             storage for BLAS.                                *
c     *                                                              *
c     *   on return                                                  *
c     *                                                              *
c     *      lftblk, array, rgtblk, work                             *
c     *             The desired decomposition of the ABD matrix.     *
c     *                                                              *
c     *             [ Note:  If iflag = -1 the matrix is exactly     *
c     *                  singular.  The factorization has been       *
c     *                  completed, but division by zero will        *
c     *                  occur if it is used to solve a system       *
c     *                  of equations.                          ]    *
c     *                                                              *
c     *      nrwblk, nbloks                                          *
c     *             Unchanged.                                       *
c     *                                                              *
c     *           b [double precision((nbloks+1)*nrwblk)]            *
c     *             The solution vector (if iflag = 0).              *
c     *                                                              *
c     *       pivot [integer((nbloks+2)*nrwblk)]                     *
c     *             The pivoting strategy.                           *
c     *                                                              *
c     *       iflag [integer]                                        *
c     *             =  0 on normal return                            *
c     *             = -1 if the ABD matrix is singular               *
c     *                                                              *
c     *             [ Note:  Only exact singularity is detected;     *
c     *                  iflag = 0 is not a guarantee of well-       *
c     *                  conditioning.  In the case where lftblk     *
c     *                  and rgtblk can be uncoupled, Lapack's       *
c     *                  DGBTRF/DGBCON may be used to obtain a       *
c     *                  condition estimate for the ABD matrix.      *
c     *                  Subroutines are included in ABDpack for     *
c     *                  transforming the slf_lu-format matrix       *
c     *                  into the correct form for input into        *
c     *                  Lapack's band routines.  See 'rotccw',      *
c     *                  'uncple' and 'mkband' for details.     ]    *
c     *                                                              *
c     *      nparts [integer]                                        *
c     *             Normally unchanged.  If, however, the            *
c     *             requested number of partitions would             *
c     *             result in fewer than 2 blocks of array(,,)       *
c     *             per partition (i.e. if nbloks < 2*nparts),       *
c     *             the subroutine automatically resets nparts       *
c     *             to 1 and uses non-partitioned SLF-LU.            *
c     *                                                              *
c     *   SUBROUTINES CALLED:                                        *
c     *                                                              *
c     *       slufa (lftblk, array, nrwblk, nbloks, rgtblk,          *
c     *              pivot, iflag, nparts, work)                     *
c     *                                                              *
c     *          Factors the ABD matrix using parallel SLF-LU.       *
c     *          Parameters are as described above.                  *
```

```
c    *                                                               *
c    *       slusl (lftblk, array, nrwblk, nbloks, rgtblk,           *
c    *              b, pivot, nparts, work)                          *
c    *                                                               *
c    *         Uses the factors returned by 'slufa' to perform       *
c    *         forward elimination and back-solve on right-hand      *
c    *         side b.  Parameters are as described above.           *
c    *                                                               *
c    *   SOLVING FOR MULTIPLE RIGHT-HAND SIDES:                      *
c    *                                                               *
c    *   'slf_lu' is called only once for a given system A x = b.    *
c    *   If iflag = 0 the system is solved.  In order to solve for   *
c    *   a different right-hand side (i.e. A x = b'), 'slusl' is     *
c    *   called directly.  The arrays lftblk, array, rgtblk, work,   *
c    *   and pivot contain the decomposition of A and pivoting       *
c    *   strategy on return from 'slf_lu' and therefore must not     *
c    *   be altered between successive calls to 'slusl'.  b is       *
c    *   the only parameter that may be changed.                     *
c    *                                                               *
c    *   REFERENCES:                                                 *
c    *                                                               *
c    *   [1]  K.R. Jackson and R.N. Pancer, The parallel solution    *
c    *        of ABD systems arising in numerical methods for        *
c    *        BVPs for ODEs, University of Toronto, Department       *
c    *        of Computer Science, Technical Report 255/91, 1992.    *
c    ***-------------------------------------------------------------***
c
         call slufa (lftblk, array, nrwblk, nbloks, rgtblk,
     *               pivot, iflag, nparts, work)
         if (iflag .eq. 0) then
            call slusl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                  b, pivot, nparts, work)
         end if
       return
       end
c-----------------------------------------------------------------------
       subroutine slufa (lftblk, array, nrwblk, nbloks, rgtblk,
     *                  pivot, iflag, nparts, work)
c
       double precision lftblk(1), array(1), rgtblk(1), work(1)
       integer nrwblk, nbloks, pivot(1), iflag, nparts
c    ***-------------------------------------------------------------***
c    *   This subroutine factors the ABD matrix defined in arrays    *
c    *   lftblk, array, and rgtblk using a variant of the parallel   *
c    *   SLF-LU algorithm.  On return, lftblk, array, rgtblk,        *
c    *   work, and pivot contain the decomposition of the matrix     *
c    *   and pivoting strategy used.  See comments in subroutine     *
c    *   'slf_lu' for further details.                               *
c    ***-------------------------------------------------------------***
         integer nsquar, wk1, wk2, wk3, minblk, remblk
c
         iflag = 0
c    ***-------------------------------------------------------------***
c    *      Use non-partitioned SLF-LU if requested number           *
```

```
c     *      of partitions would result in fewer than 2 blocks       *
c     *      per partition.                                          *
c     ***------------------------------------------------------------***
          if (nbloks .lt. 2*nparts) then
             nparts = 1
          endif
c     ***------------------------------------------------------------***
c     *      Work-space allocation:                                  *
c     *         1st/2nd-level fill-in      -   work(1)..work(wk2-1)  *
c     *         3rd-level reduced matrix   - work(wk2)..work(wk3-1)  *
c     *         temporary storage for BLAS - work(wk3)..end          *
c     *                                                             *
c     *      Total requirement:  nbloks*[(nrwblk)x(2*nrwblk)]       *
c     *                        +        [(2*nrwblk)x(2*nrwblk)]     *
c     *                        + nparts*[(2*nrwblk)x(2*nrwblk+1)]   *
c     ***------------------------------------------------------------***
          nsquar = nrwblk**2
          wk1 = 1
          wk2 = wk1 + 2*nbloks*nsquar
          wk3 = wk2 + 4*nsquar
c     ***------------------------------------------------------------***
c     *      Calculate minimum number of blocks per partition        *
c     *      Remaining blocks are distributed evenly among the       *
c     *      first partitions.                                       *
c     ***------------------------------------------------------------***
          minblk = nbloks/nparts
          remblk = nbloks - minblk*nparts
c     ***------------------------------------------------------------***
c     *      Three level factorization.                              *
c     ***------------------------------------------------------------***
          call sluf1(array,work(wk1),nrwblk,nbloks,pivot,iflag,
     *               minblk,remblk,nparts,work(wk3))
          call sluf2(array,work(wk1),nrwblk,pivot,iflag,
     *               minblk,remblk,nparts,work(wk3))
          call sluf3(lftblk,array,work(wk1),work(wk2),
     *               nrwblk,nbloks,rgtblk,pivot,iflag,work(wk3))
c     ***------------------------------------------------------------***
c     *      Set iflag to -1 if exact singularity was detected.      *
c     ***------------------------------------------------------------***
          if (iflag .ne. 0) then
             iflag = -1
          endif
       return
       end
c-----------------------------------------------------------------------
       subroutine sluf1 (array, fill, nrwblk, nbloks, pivot, iflag,
     *                   minblk, remblk, nparts, blaws)
c
       double precision array(2*nrwblk,nrwblk,1),
     *           fill(nrwblk,2*nrwblk,1), blaws(2*nrwblk,2*nrwblk+1,1)
       integer nrwblk, nbloks, pivot(1), iflag, minblk, remblk, nparts
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:            *
c     *                                                              *
```

```
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *                                                               *
c     *   In addition, the affix ' designates that the matrix is      *
c     *   transformed at the first level of the factorization.        *
c     ***-------------------------------------------------------------***
          integer ndoubl, kpart, kblok, base, top
c
          ndoubl = 2*nrwblk
c     ***-------------------------------------------------------------***
c     *      Each loop 20 iteration is independent and could          *
c     *      execute concurrently with the others.                    *
c     ***-------------------------------------------------------------***
C$DOACROSS SHARE (array, fill, nrwblk, nbloks, pivot, iflag,
C$&               minblk, remblk, nparts, blaws, ndoubl),
C$&        LOCAL (kpart, kblok, base, top)
          do 20 kpart = 1, nparts
c     ***-------------------------------------------------------------***
c     *         S_top <-- V_top                                       *
c     ***-------------------------------------------------------------***
             call partx(minblk,remblk,kpart,base,top)
             if (kpart .gt. 1) then
                call mcopy(1,nrwblk,fill(1,1,top),blaws,
     *                     blaws,array(1,1,top-1))
             else
                call mcopy(1,nrwblk,fill(1,1,top),blaws,
     *                     blaws,array(1,1,nbloks))
             endif
c     ***-------------------------------------------------------------***
c     *      SLF-LU starts at the [top; top+1] block-row pair of      *
c     *      each partition and proceeds downward sequentially        *
c     *      to the [base-1; base] block-row pair.                    *
c     ***-------------------------------------------------------------***
             do 10 kblok = top, base-1
c     ***-------------------------------------------------------------***
c     *            [W_kblok    <-- LUfact([W_kblok                    *
c     *             V_kblok+1]'           V_kblok+1]                  *
c     ***-------------------------------------------------------------***
                call DGETRF(ndoubl,nrwblk,array(1,1,kblok),ndoubl,
     *                      pivot(kblok*nrwblk+1),iflag)
c     ***-------------------------------------------------------------***
c     *            [S_kblok    <-- [L, I]^-1 P [S_kblok               *
c     *             S_kblok+1]'                  0    ]               *
c     *                                                               *
c     *      Operator [L, I]^-1 P is implemented by applying the      *
c     *      pivoting strategy recorded in pivot_kblok, followed      *
c     *      by the nrwblk Gauss transforms stored in the lower       *
c     *      trapezoid of [W_kblok; V_kblok+1]'.                      *
c     ***-------------------------------------------------------------***
                call mcopy(3,nrwblk,fill(1,1,kblok),blaws,
     *                     blaws(1,1,kpart),blaws)
```

```
               call apyLU(ndoubl,nrwblk,array(1,1,kblok),ndoubl,
     *                    pivot(kblok*nrwblk+1),blaws(1,1,kpart),
     *                    ndoubl,nrwblk,blaws(1,nrwblk+1,kpart))
               call mcopy(4,nrwblk,fill(1,1,kblok),fill(1,1,kblok+1),
     *                    blaws(1,1,kpart),blaws)
c    ***---------------------------------------------------------***
c    *            [T_kblok    <-- [L, I]^-1 P [  0                 *
c    *             W_kblok+1]'                 W_kblok+1]          *
c    *                                                             *
c    *      Operator [L, I]^-1 P is implemented as above.          *
c    ***---------------------------------------------------------***
               call mcopy(5,nrwblk,blaws,blaws,
     *                    blaws(1,1,kpart),array(1,1,kblok+1))
               call apyLU(ndoubl,nrwblk,array(1,1,kblok),ndoubl,
     *                    pivot(kblok*nrwblk+1),blaws(1,1,kpart),
     *                    ndoubl,nrwblk,blaws(1,nrwblk+1,kpart))
               call mcopy(6,nrwblk,fill(1,1,kblok),blaws,
     *                    blaws(1,1,kpart),array(1,1,kblok+1))
  10        continue
  20     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine sluf2 (array, fill, nrwblk, pivot, iflag,
     *                  minblk, remblk, nparts, blaws)
c
      double precision array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), blaws(2*nrwblk,1)
      integer nrwblk, pivot(1), iflag, minblk, remblk, nparts
c    ***---------------------------------------------------------***
c    *   The following notation is used in the comments:           *
c    *                                                             *
c    *      [W_nbloks <=> array(1,1,nbloks)                        *
c    *       V_1]                                                  *
c    *      [W_k      <=> array(1,1,k), k = 1..nbloks-1            *
c    *       V_k+1]                                                *
c    *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)          *
c    *                                                             *
c    *   In addition, the affix '/'' designates that the           *
c    *   matrix was/is transformed at the first/second level       *
c    *   of the factorization.                                     *
c    ***---------------------------------------------------------***
         integer ndoubl, kpart, base, base1, top
c    ***---------------------------------------------------------***
c    *      If there is only one partition, nothing needs to be    *
c    *      done at the second level of the factorization.         *
c    ***---------------------------------------------------------***
         if (nparts .eq. 1) return
c    ***---------------------------------------------------------***
c    *     SLF-LU starts at the [1; 2] block-row pair of the       *
c    *     second-level array and proceeds downward sequentially   *
c    *     to the [nparts-1; nparts] block-row pair.               *
c    ***---------------------------------------------------------***
         ndoubl = 2*nrwblk
```

```
         do 10 kpart = 1, nparts-1
            call partx(minblk,remblk,kpart,base,top)
            call partx(minblk,remblk,kpart+1,base1,top)
c     ***------------------------------------------------------------***
c     *            [W_base     <-- LUfact([W_base'                     *
c     *             V_base+1]''            S_base1']                  *
c     ***------------------------------------------------------------***
            call mcopy(2,nrwblk,fill(1,1,base1),blaws,
     *                 blaws,array(1,1,base))
            call DGETRF(ndoubl,nrwblk,array(1,1,base),ndoubl,
     *                  pivot(base*nrwblk+1),iflag)
c     ***------------------------------------------------------------***
c     *            [S_base    <-- [L, I]^-1 P [S_base'                 *
c     *             S_base1]''                    0    ]               *
c     *                                                                *
c     *      Operator [L, I]^-1 P is implemented by applying the       *
c     *      pivoting strategy recorded in pivot_base, followed        *
c     *      by the nrwblk Gauss transforms stored in the lower        *
c     *      trapezoid of [W_base; V_base+1]''.                        *
c     ***------------------------------------------------------------***
            call mcopy(3,nrwblk,fill(1,1,base),blaws,
     *                 blaws,blaws)
            call apyLU(ndoubl,nrwblk,array(1,1,base),ndoubl,
     *                 pivot(base*nrwblk+1),blaws,ndoubl,nrwblk,
     *                 blaws(1,nrwblk+1))
            call mcopy(4,nrwblk,fill(1,1,base),fill(1,1,base1),
     *                 blaws,blaws)
c     ***------------------------------------------------------------***
c     *            [T_base    <-- [L, I]^-1 P [  0                    *
c     *             W_base1]''                  W_base1']              *
c     *                                                                *
c     *      Operator [L, I]^-1 P is implemented as above.             *
c     ***------------------------------------------------------------***
            call mcopy(5,nrwblk,blaws,blaws,
     *                 blaws,array(1,1,base1))
            call apyLU(ndoubl,nrwblk,array(1,1,base),ndoubl,
     *                 pivot(base*nrwblk+1),blaws,ndoubl,nrwblk,
     *                 blaws(1,nrwblk+1))
            call mcopy(6,nrwblk,fill(1,1,base),blaws,
     *                 blaws,array(1,1,base1))
   10     continue
          return
          end
c-----------------------------------------------------------------------
          subroutine sluf3 (lftblk, array, fill, rdcmx,
     *                  nrwblk, nbloks, rgtblk, pivot, iflag, blaws)
c
          double precision lftblk(nrwblk,1), array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), rdcmx(2*nrwblk,1),
     *                 rgtblk(nrwblk,1), blaws(2*nrwblk,1)
          integer nrwblk, nbloks, pivot(1), iflag
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:              *
c     *                                                                *
```

```
c     *      B_a, B_b <=> lftblk, rgtblk                              *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *                                                               *
c     *   In addition, the affix ''/''' designates that the           *
c     *   matrix was/is transformed at the second/third level         *
c     *   of the factorization.                                       *
c     ***-----------------------------------------------------------***
          integer i, j, ndoubl
c     ***-----------------------------------------------------------***
c     *      The third-level block-array is of the form               *
c     *                                                               *
c     *            [   B_a        B_b                                *
c     *             S_nbloks'' W_nbloks'']                            *
c     *                                                               *
c     *      The LU-factorization of this block-array is stored       *
c     *      in the rdcmx(,) work-space.  The pivot indices are       *
c     *      stored in pivot(nbloks*nrwblk+1..(nbloks+2)*nrwblk).     *
c     ***-----------------------------------------------------------***
          ndoubl = 2*nrwblk
          do 20 j = 1, nrwblk
             do 10 i = 1, nrwblk
                rdcmx(i,j) = lftblk(i,j)
                rdcmx(i,nrwblk+j) = rgtblk(i,j)
                rdcmx(nrwblk+i,j) = fill(i,j,nbloks)
                rdcmx(nrwblk+i,nrwblk+j) = array(i,j,nbloks)
  10         continue
  20      continue
          call DGETRF(ndoubl,ndoubl,rdcmx,ndoubl,
     *                pivot(nbloks*nrwblk+1),iflag)
          return
          end
c-----------------------------------------------------------------------
          subroutine slusl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                  b, pivot, nparts, work)
c
          double precision lftblk(1), array(1), rgtblk(1), b(1), work(1)
          integer nrwblk, nbloks, pivot(1), nparts
c     ***-----------------------------------------------------------***
c     *   Given the factors of ABD matrix A computed by subroutine    *
c     *   'slufa' and stored in arrays lftblk, array, rgtblk,         *
c     *   work and pivot, this subroutine solves the linear system    *
c     *   A x = b.  b is overwritten with x.  See comments in         *
c     *   subroutine 'slf_lu' for further details.                    *
c     ***-----------------------------------------------------------***
          integer nsquar, wk1, wk2, wk3, minblk, remblk
c     ***-----------------------------------------------------------***
c     *      Work-space allocation:                                   *
c     *         1st/2nd-level fill-in      -   work(1)..work(wk2-1)   *
c     *         3rd-level reduced matrix   - work(wk2)..work(wk3-1)   *
c     *         temporary storage for BLAS - work(wk3)..end           *
```

```
c     *                                                               *
c     *      Total requirement:  nbloks*[(nrwblk)x(2*nrwblk)]         *
c     *                        +        [(2*nrwblk)x(2*nrwblk)]       *
c     *                        + nparts*[(2*nrwblk)x(2*nrwblk+1)]     *
c     ***-------------------------------------------------------------***
          nsquar = nrwblk**2
          wk1 = 1
          wk2 = wk1 + 2*nbloks*nsquar
          wk3 = wk2 + 4*nsquar
c     ***-------------------------------------------------------------***
c     *      Calculate minimum number of blocks per partition         *
c     *      Remaining blocks are distributed evenly among the        *
c     *      first partitions.                                        *
c     ***-------------------------------------------------------------***
          minblk = nbloks/nparts
          remblk = nbloks - minblk*nparts
c     ***-------------------------------------------------------------***
c     *      Three level forward elimination.                         *
c     ***-------------------------------------------------------------***
          call slusf1(array,nrwblk,b,pivot,
     *               minblk,remblk,nparts,work(wk3))
          call slusf2(array,nrwblk,b,pivot,
     *               minblk,remblk,nparts,work(wk3))
          call slusf3(work(wk2),nrwblk,nbloks,b,b,pivot,work(wk3))
c     ***-------------------------------------------------------------***
c     *      Three level back-solve.                                  *
c     ***-------------------------------------------------------------***
          call slusb3(work(wk2),nrwblk,nbloks,b,b,work(wk3))
          call slusb2(array,work(wk1),nrwblk,nbloks,b,
     *               minblk,remblk,nparts,work(wk3))
          call slusb1(array,work(wk1),nrwblk,b,
     *               minblk,remblk,nparts,work(wk3))
      return
      end
c-----------------------------------------------------------------------
      subroutine slusf1 (array, nrwblk, phi, pivot,
     *                   minblk, remblk, nparts, blaws)
c
      double precision array(2*nrwblk,nrwblk,1), phi(1),
     *                 blaws(2*nrwblk,2*nrwblk+1,1)
      integer nrwblk, pivot(1), minblk, remblk, nparts
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *          phi_k <=> phi(k*nrwblk+1)                            *
c     *                                                               *
c     *   In addition, the affix ' designates that the vector is      *
c     *   transformed at the first level of the forward solve.        *
c     ***-------------------------------------------------------------***
          integer ndoubl, kpart, kblok, base, top
```

```
c
         ndoubl = 2*nrwblk
c    ***---------------------------------------------------------***
c    *      Each loop 20 iteration is independent and could         *
c    *      execute concurrently with the others.                   *
c    ***---------------------------------------------------------***
C$DOACROSS SHARE (array, nrwblk, phi, pivot,
C$&              minblk, remblk, nparts, blaws, ndoubl),
C$&        LOCAL (kpart, kblok, base, top)
         do 20 kpart = 1, nparts
c    ***---------------------------------------------------------***
c    *      Forward elimination starts at [phi_top; phi_top+1]      *
c    *      of each partition and proceeds downward sequentially    *
c    *      to [phi_base-1; phi_base].                              *
c    ***---------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            do 10 kblok = top, base-1
c    ***---------------------------------------------------------***
c    *          [phi_kblok    <-- [L, I]^-1 P [phi_kblok           *
c    *           phi_kblok+1]'                 phi_kblok+1]        *
c    *                                                             *
c    *      Operator [L, I]^-1 P is implemented by applying the    *
c    *      pivoting strategy recorded in pivot_kblok, followed    *
c    *      by the nrwblk Gauss transforms stored in the lower     *
c    *      trapezoid of [W_kblok; V_kblok+1]'.                    *
c    ***---------------------------------------------------------***
               call apyLU(ndoubl,nrwblk,array(1,1,kblok),ndoubl,
     *                    pivot(kblok*nrwblk+1),phi(kblok*nrwblk+1),
     *                    ndoubl,1,blaws(1,nrwblk+1,kpart))
  10        continue
  20     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine slusf2 (array, nrwblk, phi, pivot,
     *                   minblk, remblk, nparts, blaws)
c
      double precision array(2*nrwblk,nrwblk,1), phi(1),
     *                 blaws(2*nrwblk,1)
      integer nrwblk, pivot(1), minblk, remblk, nparts
c    ***---------------------------------------------------------***
c    *   The following notation is used in the comments:           *
c    *                                                             *
c    *      [W_nbloks <=> array(1,1,nbloks)                        *
c    *       V_1]                                                  *
c    *      [W_k      <=> array(1,1,k), k = 1..nbloks-1            *
c    *       V_k+1]                                                *
c    *          phi_k <=> phi(k*nrwblk+1)                          *
c    *                                                             *
c    *   In addition, the affix '/'' designates that the           *
c    *   vector was/is transformed at the first/second level       *
c    *   of the forward solve.                                     *
c    ***---------------------------------------------------------***
         integer ndoubl, kpart, base, base1, top
```

```
c    ***-------------------------------------------------------------***
c    *      If there is only one partition, nothing needs to be     *
c    *      done at the second level of the forward solve.          *
c    ***-------------------------------------------------------------***
         if (nparts .eq. 1) return
c    ***-------------------------------------------------------------***
c    *    Forward elimination starts at [phi_base_1; phi_base_2]    *
c    *    of the second-level system and proceeds downward          *
c    *    sequentially to [phi_base_nparts-1; phi_base_nparts].     *
c    ***-------------------------------------------------------------***
         ndoubl = 2*nrwblk
         do 10 kpart = 1, nparts-1
            call partx(minblk,remblk,kpart,base,top)
            call partx(minblk,remblk,kpart+1,base1,top)
c    ***-------------------------------------------------------------***
c    *          [phi_base    <-- [L, I]^-1 P [phi_base''            *
c    *           phi_base1]''                 phi_base1']           *
c    *                                                              *
c    *      Operator [L, I]^-1 P is implemented by applying the     *
c    *      pivoting strategy recorded in pivot_base, followed      *
c    *      by the nrwblk Gauss transforms stored in the lower      *
c    *      trapezoid of [W_base; V_base+1]''.                      *
c    ***-------------------------------------------------------------***
            call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
            call DCOPY(nrwblk,phi(base1*nrwblk+1),1,
     *                        blaws(nrwblk+1,1),1)
            call apyLU(ndoubl,nrwblk,array(1,1,base),ndoubl,
     *                 pivot(base*nrwblk+1),blaws,ndoubl,1,
     *                 blaws(1,nrwblk+1))
            call DCOPY(nrwblk,blaws,1,phi(base*nrwblk+1),1)
            call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                        phi(base1*nrwblk+1),1)
   10    continue
         return
         end
c--------------------------------------------------------------------
         subroutine slusf3 (rdcmx, nrwblk, nbloks, beta, phi,
     *                   pivot, blaws)
c
         double precision rdcmx(2*nrwblk,1), beta(1),
     *                  phi(1), blaws(2*nrwblk,1)
         integer nrwblk, nbloks, pivot(1)
c    ***-------------------------------------------------------------***
c    *   The following notation is used in the comments:            *
c    *                                                              *
c    *          beta <=> beta(1) (<=> phi_0)                        *
c    *         phi_k <=> phi(k*nrwblk+1), k >= 1                    *
c    *                                                              *
c    *   In addition, the affix ''/''' designates that the          *
c    *   vector was/is transformed at the second/third level        *
c    *   of the forward solve.                                      *
c    ***-------------------------------------------------------------***
         integer ndoubl
c    ***-------------------------------------------------------------***
```

```
c     *      The right-hand side of the third-level system is        *
c     *      transformed as follows:                                 *
c     *                                                              *
c     *        [beta          <-- [L, I]^-1 P [beta                  *
c     *         phi_nbloks]'''                 phi_nbloks'']         *
c     *                                                              *
c     *      Operator [L, I]^-1 P is implemented by applying the     *
c     *      2*nrwblk pivoting strategy recorded in pivot_nbloks,    *
c     *      followed by the 2*nrwblk Gauss transforms stored in     *
c     *      the lower triangle of rdcmx(,).                         *
c     ***------------------------------------------------------------***
          ndoubl = 2*nrwblk
          call DCOPY(nrwblk,beta,1,blaws,1)
          call DCOPY(nrwblk,phi(nbloks*nrwblk+1),1,
     *                      blaws(nrwblk+1,1),1)
          call apyLU(ndoubl,ndoubl,rdcmx,ndoubl,
     *               pivot(nbloks*nrwblk+1),blaws,ndoubl,1,
     *               blaws(1,nrwblk+1))
          call DCOPY(nrwblk,blaws,1,beta,1)
          call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                      phi(nbloks*nrwblk+1),1)
          return
          end
c-----------------------------------------------------------------------
          subroutine slusb3 (rdcmx, nrwblk, nbloks, beta, phi, blaws)
c
          double precision rdcmx(2*nrwblk,1), beta(1),
     *                 phi(1), blaws(2*nrwblk,1)
          integer nrwblk, nbloks
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:            *
c     *                                                              *
c     *          beta <=> beta(1) (<=> phi_0)                        *
c     *         phi_k <=> phi(k*nrwblk+1), k >= 1                    *
c     *                                                              *
c     *   Since [beta; phi] is overwritten with the solution,        *
c     *                                                              *
c     *      y_a <=> y_0      <=> beta                               *
c     *              y_k      <=> phi_k,  k = 1, nbloks-1            *
c     *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)               *
c     *                                                              *
c     *   In addition, the affix ''' designates that the solution    *
c     *   vector is obtained at the third level of the back-solve.   *
c     ***------------------------------------------------------------***
          integer ndoubl
c     ***------------------------------------------------------------***
c     *      y_a''' and y_b''' are obtained by solving the upper-    *
c     *      triangular third-level system                           *
c     *                                                              *
c     *         rdcmx(,) [y_a  = [beta                               *
c     *                   y_b]    phi_nbloks]                       *
c     ***------------------------------------------------------------***
          ndoubl = 2*nrwblk
          call DCOPY(nrwblk,beta,1,blaws,1)
```

```
         call DCOPY(nrwblk,phi(nbloks*nrwblk+1),1,
     *                     blaws(nrwblk+1,1),1)
         call DTRSM('L','U','N','N',ndoubl,1,1.d0,
     *              rdcmx,ndoubl,blaws,ndoubl)
         call DCOPY(nrwblk,blaws,1,beta,1)
         call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                     phi(nbloks*nrwblk+1),1)
       return
       end
c-----------------------------------------------------------------------
       subroutine slusb2 (array, fill, nrwblk, nbloks, phi,
     *                    minblk, remblk, nparts, blaws)
c
       double precision array(2*nrwblk,nrwblk,1),
     *             fill(nrwblk,2*nrwblk,1), phi(1), blaws(2*nrwblk,1)
       integer nrwblk, nbloks, minblk, remblk, nparts
c      ***-------------------------------------------------------------***
c      *   The following notation is used in the comments:               *
c      *                                                                 *
c      *      [W_nbloks <=> array(1,1,nbloks)                            *
c      *       V_1]                                                      *
c      *      [W_k      <=> array(1,1,k), k = 1..nbloks-1                *
c      *       V_k+1]                                                    *
c      *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)              *
c      *          phi_k <=> phi(k*nrwblk+1)                              *
c      *                                                                 *
c      *   Since phi is overwritten with the solution,                   *
c      *                                                                 *
c      *      y_a <=> y_0      <=> phi_0  (<=> beta in slusb3.f)         *
c      *              y_k      <=> phi_k,  k = 1, nbloks-1               *
c      *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)                  *
c      *                                                                 *
c      *   In addition, the affix '''/'' designates that the             *
c      *   solution vector was/is obtained at the third/second           *
c      *   level of the back-solve.                                      *
c      ***-------------------------------------------------------------***
         integer ndoubl, kpart, base, basep, top
c      ***-------------------------------------------------------------***
c      *      If there is only one partition, nothing needs to be        *
c      *      done at the second level of the back-solve.                *
c      ***-------------------------------------------------------------***
         if (nparts .eq. 1) return
c      ***-------------------------------------------------------------***
c      *      Back-solve starts at the second-last block-row of the      *
c      *      second-level system and proceeds upward sequentially       *
c      *      to the first block-row.                                    *
c      ***-------------------------------------------------------------***
         ndoubl = 2*nrwblk
         base = nbloks
         call DCOPY(nrwblk,phi,1,blaws,1)
         do 10 kpart = nparts-1, 1, -1
            basep = base
            call partx(minblk,remblk,kpart,base,top)
c      ***-------------------------------------------------------------***
```

```
c     *   phi_base <-- phi_base - S_base y_0''' - T_base y_basep''   *
c     ***-------------------------------------------------------------***
            call DCOPY(nrwblk,phi(basep*nrwblk+1),1,
     *                        blaws(nrwblk+1,1),1)
            call DGEMV('N',nrwblk,ndoubl,
     *                 -1.d0,fill(1,1,base),nrwblk,
     *                 blaws,1,1.d0,phi(base*nrwblk+1),1)
c     ***-------------------------------------------------------------***
c     *   y_base'' is now obtained by solving the upper-triangular   *
c     *   nrwblkxnrwblk system  W_base y_base = phi_base             *
c     ***-------------------------------------------------------------***
            call DTRSM('L','U','N','N',nrwblk,1,1.d0,
     *                 array(1,1,base),ndoubl,
     *                 phi(base*nrwblk+1),nrwblk)
  10     continue
         return
         end
c-----------------------------------------------------------------------
         subroutine slusb1 (array, fill, nrwblk, phi,
     *                   minblk, remblk, nparts, blaws)
c
         double precision array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), phi(1),
     *                 blaws(2*nrwblk,nrwblk+1,1)
         integer nrwblk, minblk, remblk, nparts
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *          phi_k <=> phi(k*nrwblk+1)                            *
c     *                                                               *
c     *   Since phi is overwritten with the solution,                 *
c     *                                                               *
c     *      y_a <=> y_0      <=> phi_0  (<=> beta in slusb3.f)       *
c     *              y_k      <=> phi_k,  k = 1, nbloks-1             *
c     *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)                *
c     *                                                               *
c     *   In addition, the affix ''/'' designates that the            *
c     *   solution vector was/is obtained at the second/first         *
c     *   level of the back-solve.                                    *
c     ***-------------------------------------------------------------***
            integer ndoubl, kpart, kblok, base, top
c
            ndoubl = 2*nrwblk
c     ***-------------------------------------------------------------***
c     *      Each loop 20 iteration is independent and could          *
c     *      execute concurrently with the others.                    *
c     ***-------------------------------------------------------------***
C$DOACROSS SHARE (array, fill, nrwblk, phi,
C$&               minblk, remblk, nparts, blaws, ndoubl),
```

```
C$&        LOCAL (kpart, kblok, base, top)
         do 20 kpart = 1, nparts
c     ***-------------------------------------------------------------***
c     *      Back-solve starts at the second-last block-row of        *
c     *      each partition and proceeds upward sequentially to       *
c     *      the first block-row.                                     *
c     ***-------------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            call DCOPY(nrwblk,phi((top-1)*nrwblk+1),1,
     *                        blaws(1,1,kpart),1)
            do 10 kblok = base-1, top, -1
c     ***-------------------------------------------------------------***
c     *         phi_kblok <-- phi_kblok - S_kblok y_top-1''           *
c     *                                - T_kblok y_kblok+1'           *
c     ***-------------------------------------------------------------***
               call DCOPY(nrwblk,phi((kblok+1)*nrwblk+1),1,
     *                           blaws(nrwblk+1,1,kpart),1)
               call DGEMV('N',nrwblk,ndoubl,
     *                    -1.d0,fill(1,1,kblok),nrwblk,
     *                    blaws(1,1,kpart),1,1.d0,
     *                    phi(kblok*nrwblk+1),1)
c     ***-------------------------------------------------------------***
c     *   y_kblok' is now obtained by solving the upper-triangular    *
c     *   nrwblkxnrwblk system  W_kblok y_kblok = phi_kblok           *
c     ***-------------------------------------------------------------***
               call DTRSM('L','U','N','N',nrwblk,1,1.d0,
     *                    array(1,1,kblok),ndoubl,
     *                    phi(kblok*nrwblk+1),nrwblk)
  10        continue
  20     continue
       return
       end
c-----------------------------------------------------------------------
       subroutine apyLU (m, n, LU, ldLU, ipiv, C, ldC, nC, blaws)
c
       double precision LU(ldLU,1), C(ldC,1), blaws(n,1)
       integer m, n, ldLU, ipiv(1), ldC, nC
c     ***-------------------------------------------------------------***
c     *   Given an LU factorization computed by Lapack's DGETRF,      *
c     *   'apyLU' applies the pivoting strategy recorded in ipiv      *
c     *   and Gauss transforms stored in the lower trapezoid of       *
c     *   LU to the m x nC matrix C.                                  *
c     *                                                               *
c     *    on entry                                                   *
c     *                                                               *
c     *           m [integer]                                         *
c     *             The number of rows in LU and C.                   *
c     *                                                               *
c     *           n [integer]                                         *
c     *             The number of columns in LU.  n <= m.             *
c     *                                                               *
c     *          LU [double precision(ldLU,n)]                        *
c     *             The factors L and U computed and stored           *
c     *             by Lapack's DGETRF.                               *
```

```
c     *                                                               *
c     *        ldLU [integer]                                         *
c     *             The leading dimension of the array LU.            *
c     *                                                               *
c     *        ipiv [integer(n)]                                      *
c     *             The pivot indices computed and stored             *
c     *             by Lapack's DGETRF.                               *
c     *                                                               *
c     *           C [double precision(ldC,nC)]                        *
c     *             The matrix to be transformed.                     *
c     *                                                               *
c     *         ldC [integer]                                         *
c     *             The leading dimension of the array C.             *
c     *                                                               *
c     *          nC [integer]                                         *
c     *             The number of columns in C.  nC <= n.             *
c     *                                                               *
c     *       blaws [double precision(n,2*n)]                         *
c     *             Work space.                                       *
c     *                                                               *
c     *   on return                                                   *
c     *                                                               *
c     *           C [double precision(ldC,nC)]                        *
c     *             The transformed matrix.                           *
c     *                                                               *
c     *             All other parameters are unchanged.               *
c     ***-------------------------------------------------------------***
          integer i, j
c     ***-------------------------------------------------------------***
c     *      Apply the pivoting strategy to C.                        *
c     ***-------------------------------------------------------------***
          call DLASWP(nC,C,ldC,1,n,ipiv,1)
c     ***-------------------------------------------------------------***
c     *    The Gauss transforms are stored in [L; A], the lower       *
c     *    trapezoid of LU, where L is n x n unit lower triangular    *
c     *    and A is k x n, k = m - n.  The transforms are applied     *
c     *    by solving [L O; A I] [C1'; C2'] = [C1; C2] where O is     *
c     *    the n x k zero matrix, I is the k x k identity matrix,     *
c     *    and C1 and C2 are n x nC and k x nC, respectively.         *
c     *                                                               *
c     *    First, solve L C1' = C1.  If n = m, this is enough.        *
c     *    If n < m, C2' is computed from C1' as described below.     *
c     ***-------------------------------------------------------------***
          call DTRSM('L','L','N','U',n,nC,1.d0,LU,ldLU,C,ldC)
          if (n .eq. m) return
c     ***-------------------------------------------------------------***
c     *      Repack A into consecutive work space.                    *
c     ***-------------------------------------------------------------***
          do 20 j = 1, n
             do 10 i = 1, n
                blaws(i,j) = LU(n+i,j)
  10         continue
  20      continue
          if (nC .gt. 1) then
```

```
c    ***-------------------------------------------------------------***
c    *       Compute C2' = C2 - A*C1' using DGEMM.  C2 first        *
c    *       must be repacked into consecutive work space.          *
c    ***-------------------------------------------------------------***
           do 40 j = 1, nC
              do 30 i = 1, n
                 blaws(i,n+j) = C(n+i,j)
  30         continue
  40       continue
           call DGEMM('N','N',n,nC,n,-1.d0,blaws,n,
     *                C,ldC,1.d0,blaws(1,n+1),n)
           do 60 j = 1, nC
              do 50 i = 1, n
                 C(n+i,j) = blaws(i,n+j)
  50         continue
  60       continue
        else
c    ***-------------------------------------------------------------***
c    *       Compute C2' = C2 - A*C1' using DGEMV.                  *
c    ***-------------------------------------------------------------***
           call DGEMV('N',n,n,-1.d0,blaws,n,C,1,1.d0,C(n+1,1),1)
        end if
      return
      end
c-----------------------------------------------------------------------
      subroutine mcopy (sel, n, C, D, E, F)
c
      double precision C(n,1), D(n,1), E(2*n,1), F(2*n,1)
      integer sel, n
c    ***-------------------------------------------------------------***
c    *   'mcopy' performs the matrix copy selected by sel.          *
c    *   Note:  Due to the nature of the storage organization,      *
c    *   vectorized copying (Lapack's DCOPY) is not possible.       *
c    *                                                              *
c    *    on entry                                                  *
c    *                                                              *
c    *         sel [integer]                                        *
c    *             Copy selection (see below for details).          *
c    *                                                              *
c    *           n [integer]                                        *
c    *             The number of rows in C and D and the number     *
c    *             of columns in E and F.  There are 2*n columns    *
c    *             in C and D and 2*n rows in E and F.              *
c    *                                                              *
c    *  C, D, E, F [double precision                                *
c    *              (n,2*n), (n,2*n), (2*n,n), (2*n,n)]             *
c    *             The matrices before copying.                     *
c    *                                                              *
c    *   on return                                                  *
c    *                                                              *
c    *         sel [integer]                                        *
c    *             Unchanged.                                       *
c    *                                                              *
c    *           n [integer]                                        *
```

```
c    *             Unchanged.                                          *
c    *                                                                 *
c    *  C, D, E, F [double precision                                   *
c    *              (n,2*n), (n,2*n), (2*n,n), (2*n,n)]                *
c    *             The matrices after copying.                         *
c    ***---------------------------------------------------------------***
         integer i, j, j1
c    ***---------------------------------------------------------------***
c    *      The following notation is used in the comments:            *
c    *                                                                 *
c    *             C <=> [C1 C2]    D <=> [D1 D2]                     *
c    *             E <=> [E1        F <=> [F1                         *
c    *                    E2]              F2]                        *
c    *                                                                 *
c    *      Ck, Dk, Ek and Fk, k = 1, 2, are each nxn matrices.        *
c    *      (0) is the nxn zero matrix.                                *
c    ***---------------------------------------------------------------***
         go to (100, 200, 300, 400, 500, 600) sel
c    ***---------------------------------------------------------------***
c    *         Copy selection #1:  C1 <- F2                            *
c    ***---------------------------------------------------------------***
 100        do 120 j = 1, n
               do 110 i = 1, n
                  C(i,j) = F(n+i,j)
 110           continue
 120        continue
         return
c    ***---------------------------------------------------------------***
c    *         Copy selection #2:  F2 <- C1                            *
c    ***---------------------------------------------------------------***
 200        do 220 j = 1, n
               do 210 i = 1, n
                  F(n+i,j) = C(i,j)
 210           continue
 220        continue
         return
c    ***---------------------------------------------------------------***
c    *         Copy selection #3:  E1 <- C1                            *
c    *                             E2 <- (0)                           *
c    ***---------------------------------------------------------------***
 300        do 320 j = 1, n
               do 310 i = 1, n
                  E(i,j) = C(i,j)
                  E(n+i,j) = 0.d0
 310           continue
 320        continue
         return
c    ***---------------------------------------------------------------***
c    *         Copy selection #4:  C1 <- E1                            *
c    *                             D1 <- E2                            *
c    ***---------------------------------------------------------------***
 400        do 420 j = 1, n
               do 410 i = 1, n
                  C(i,j) = E(i,j)
```

```
                  D(i,j) = E(n+i,j)
 410           continue
 420        continue
        return
c    ***---------------------------------------------------------------***
c    *         Copy selection #5:  E1 <- (0)                           *
c    *                             E2 <- F1                            *
c    ***---------------------------------------------------------------***
 500        do 520 j = 1, n
               do 510 i = 1, n
                  E(i,j) = 0.d0
                  E(n+i,j) = F(i,j)
 510           continue
 520        continue
        return
c    ***---------------------------------------------------------------***
c    *         Copy selection #6:  C2 <- E1                            *
c    *                             F1 <- E2                            *
c    ***---------------------------------------------------------------***
 600        do 620 j = 1, n
               j1 = n + j
               do 610 i = 1, n
                  C(i,j1) = E(i,j)
                  F(i,j) = E(n+i,j)
 610           continue
 620        continue
        return
      end
c-----------------------------------------------------------------------
      subroutine partx (minblk, remblk, k, base, top)
c
      integer minblk, remblk, k, base, top
c    ***---------------------------------------------------------------***
c    *   'partx' calculates the index of the base and top block      *
c    *   of the k-th partition.  The indexing scheme assumes that    *
c    *   nbloks >= 2*nparts, so minblk >= 2 and remblk >= 0.         *
c    *                                                               *
c    *    on entry                                                   *
c    *                                                               *
c    *      minblk [integer]                                         *
c    *             minimum number of blocks/partition                *
c    *             (minblk = nbloks/nparts)                          *
c    *                                                               *
c    *      remblk [integer]                                         *
c    *             first remblk partitions have minblk+1 blocks      *
c    *             (remblk = nbloks - minblk*nparts)                 *
c    *                                                               *
c    *           k [integer]                                         *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *        base [integer]                                         *
c    *             index of base block of k-th partition             *
c    *                                                               *
```

```
c    *         top [integer]                                          *
c    *             index of top block of k-th partition               *
c    ***------------------------------------------------------------***
         integer khigh, klow
c
         if (k .le. remblk) then
            khigh = k
            klow = 0
         else
            khigh = remblk
            klow = k - khigh
         endif
         base = khigh*(minblk+1) + klow*minblk
         top = base - minblk + 1
         if (k .le. remblk) then
            top = top - 1
         endif
       return
       end
```

E.3 *SLF*-QR

```
      subroutine slf_qr (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, tau, iflag, nparts, work)
c
      double precision lftblk(1), array(1), rgtblk(1), b(1),
     *                 tau(1), work(1)
      integer nrwblk, nbloks, iflag, nparts
c     ***------------------------------------------------------------***
c     *   This subroutine solves the linear system A x = b where     *
c     *   A is an Almost Block Diagonal matrix of the form           *
c     *                                                              *
c     *             lftblk                    rgtblk               *
c     *                     a                                        *
c     *             bgnblk  r                                        *
c     *                     r                                        *
c     *                     a   a                                    *
c     *                     y   r                                    *
c     *                   (,,1) r                                    *
c     *                         a   .                                *
c     *                         y     .                              *
c     *                       (,,2)     .                            *
c     *                                   a                          *
c     *                                   r                          *
c     *                                   r                          *
c     *                                   a                          *
c     *                                   y   endblk                 *
c     *                          (,,nbloks-1)                        *
c     *                                                              *
c     *   lftblk and rgtblk are each nrwblkxnrwblk, array(,,k)       *
c     *   is 2*nrwblkxnrwblk, k = 1..nbloks, endblk/bgnblk alias     *
c     *   the upper/lower nrwblkxnrwblk block of array(,,nbloks),    *
c     *   bgnblk overlaps the first nrwblk rows of array(,,1),       *
c     *   {[array(,,k) array(,,k+1)], k = 1..nbloks-2} overlap       *
c     *   by nrwblk rows each, and endblk overlaps the last nrwblk   *
c     *   rows of array(,,nbloks-1).  The linear system is square    *
c     *   and of order (nbloks+1)*nrwblk.                            *
c     *                                                              *
c     *  [ Note:  ABDs often arise in other forms.  For example,     *
c     *       lftblk and rgtblk may be uncoupled so that lftblk      *
c     *       appears at the top of the matrix and rgtblk appears    *
c     *       at the bottom.  Also, the blocks in array(,,) are      *
c     *       often arranged so that array(,,k) holds the left       *
c     *       and right blocks in block-row k.  In these cases,      *
c     *       the ABD system first can be transformed into the       *
c     *       correct form for input to 'slf_qr' using auxiliary     *
c     *       routines included with this package.  See 'couple'     *
c     *       and 'rotcw' for details.  Alternatively, 'slfqrc',     *
c     *       a modified version of 'slf_qr' that incorporates       *
c     *       both 'couple' and 'rotcw', can be used.           ]    *
c     *                                                              *
c     *   THE ALGORITHM:                                             *
c     *                                                              *
c     *   The system is decomposed and solved using a variant        *
```

```
c     *   of the parallel SLF-QR algorithm described in [1].         *
c     *   Parallelism is achieved by slicing the system into         *
c     *   'nparts' partitions in such a way that each partition      *
c     *   can be processed independently.  Assuming at least one     *
c     *   processor is available per partition, a speed-up of S      *
c     *   (over sequential SLF-QR) may be attained where             *
c     *                                                              *
c     *            1 <= S < nparts,                                  *
c     *                                                              *
c     *        with  S = 1       if  nbloks < 2*nparts,              *
c     *         and  S ~ nparts  if  nparts << nbloks/nparts.        *
c     *                                                              *
c     *   In other words, for systems of sufficiently high order,    *
c     *   speed-up is approximately linear with respect to nparts    *
c     *   when nparts is sufficiently small.  Sample problems and    *
c     *   timing benchmarks are included with this package.          *
c     *                                                              *
c     *   PARAMETERS:                                                *
c     *                                                              *
c     *    on entry                                                  *
c     *                                                              *
c     *      lftblk [double precision(nrwblk,nrwblk)]                *
c     *             The top left block of the ABD matrix.            *
c     *                                                              *
c     *       array [double precision(2*nrwblk,nrwblk,nbloks)]       *
c     *             array(,,k), k = 1..nbloks-1, contains the        *
c     *             k-th 2*nrwblkxnrwblk block of the ABD matrix     *
c     *             as described above.  array(,,nbloks) contains    *
c     *             endblk/bgnblk as described above.                *
c     *                                                              *
c     *      nrwblk [integer]                                        *
c     *             The number of columns in lftblk, array(,,k)      *
c     *             and rgtblk.  The number of rows in               *
c     *             lftblk and rgtblk.  There are 2*nrwblk           *
c     *             rows in array(,,k).                              *
c     *                                                              *
c     *      nbloks [integer]                                        *
c     *             The number of 2*nrwblkxnrwblk blocks             *
c     *             in array(,,).                                    *
c     *                                                              *
c     *      rgtblk [double precision(nrwblk,nrwblk)]                *
c     *             The top right block of the ABD matrix.           *
c     *                                                              *
c     *           b [double precision((nbloks+1)*nrwblk)]            *
c     *             The right-hand side vector.                      *
c     *                                                              *
c     *         tau [double precision((nbloks+1)*nrwblk)]            *
c     *             Work space to hold the scalar factors of         *
c     *             the elementary reflectors used to compute        *
c     *             the decomposition.                               *
c     *                                                              *
c     *      nparts [integer]                                        *
c     *             The number of partitions to use in the           *
c     *             decomposition and solve.                         *
```

```
c     *                                                               *
c     *        work [double precision(2*nparts*nrwblk +               *
c     *                    (2*nbloks+2*nparts+4)*nrwblk**2)]          *
c     *             Work space to hold fill-in and local              *
c     *             storage for BLAS.                                 *
c     *                                                               *
c     *   on return                                                   *
c     *                                                               *
c     *      lftblk, array, rgtblk, work                              *
c     *             The desired decomposition of the ABD matrix.      *
c     *                                                               *
c     *             [ Note:  If iflag = -1 the matrix is exactly      *
c     *                  singular.  The factorization has been        *
c     *                  completed, but division by zero will         *
c     *                  occur if it is used to solve a system        *
c     *                  of equations.                          ]     *
c     *                                                               *
c     *      nrwblk, nbloks                                           *
c     *             Unchanged.                                        *
c     *                                                               *
c     *           b [double precision((nbloks+1)*nrwblk)]             *
c     *             The solution vector (if iflag = 0).               *
c     *                                                               *
c     *          tau [double precision((nbloks+1)*nrwblk)]            *
c     *             The scalar factors of the elementary              *
c     *             reflectors.                                       *
c     *                                                               *
c     *        iflag [integer]                                        *
c     *             =  0 on normal return                             *
c     *             = -1 if the ABD matrix is singular                *
c     *                                                               *
c     *             [ Note:  Only exact singularity is detected;      *
c     *                  iflag = 0 is not a guarantee of well-        *
c     *                  conditioning.  In the case where lftblk      *
c     *                  and rgtblk can be uncoupled, Lapack's        *
c     *                  DGBTRF/DGBCON may be used to obtain a        *
c     *                  condition estimate for the ABD matrix.       *
c     *                  Subroutines are included in ABDpack for      *
c     *                  transforming the slf_qr-format matrix        *
c     *                  into the correct form for input into         *
c     *                  Lapack's band routines.  See 'rotccw',       *
c     *                  'uncple' and 'mkband' for details.     ]     *
c     *                                                               *
c     *      nparts [integer]                                         *
c     *             Normally unchanged.  If, however, the             *
c     *             requested number of partitions would              *
c     *             result in fewer than 2 blocks of array(,,)        *
c     *             per partition (i.e. if nbloks < 2*nparts),        *
c     *             the subroutine automatically resets nparts        *
c     *             to 1 and uses non-partitioned SLF-QR.             *
c     *                                                               *
c     *   SUBROUTINES CALLED:                                         *
c     *                                                               *
c     *       sqrfa (lftblk, array, nrwblk, nbloks, rgtblk,           *
```

```
c     *              tau, iflag, nparts, work)                        *
c     *                                                               *
c     *          Factors the ABD matrix using parallel SLF-QR.        *
c     *          Parameters are as described above.                   *
c     *                                                               *
c     *       sqrsl (lftblk, array, nrwblk, nbloks, rgtblk,           *
c     *              b, tau, nparts, work)                            *
c     *                                                               *
c     *          Uses the factors returned by 'sqrfa' to perform      *
c     *          forward elimination and back-solve on right-hand     *
c     *          side b.  Parameters are as described above.          *
c     *                                                               *
c     *   SOLVING FOR MULTIPLE RIGHT-HAND SIDES:                      *
c     *                                                               *
c     *   'slf_qr' is called only once for a given system A x = b.    *
c     *   If iflag = 0 the system is solved.  In order to solve       *
c     *   for a different right-hand side (i.e. A x = b'), 'sqrsl'    *
c     *   is called directly.  The arrays lftblk, array, rgtblk,      *
c     *   work, and tau contain the decomposition of A and scalar     *
c     *   factors of the elementary reflectors used to compute        *
c     *   the decompostion on return from 'slf_qr' and therefore      *
c     *   must not be altered between successive calls to 'sqrsl'.    *
c     *   b is the only parameter that may be changed.                *
c     *                                                               *
c     *   REFERENCES:                                                 *
c     *                                                               *
c     *   [1]  K.R. Jackson and R.N. Pancer, The parallel solution    *
c     *        of ABD systems arising in numerical methods for        *
c     *        BVPs for ODEs, University of Toronto, Department       *
c     *        of Computer Science, Technical Report 255/91, 1992.    *
c     ***-----------------------------------------------------------***
c
          call sqrfa (lftblk, array, nrwblk, nbloks, rgtblk,
     *                tau, iflag, nparts, work)
          if (iflag .eq. 0) then
             call sqrsl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, tau, nparts, work)
          end if
       return
       end
c---------------------------------------------------------------------
       subroutine sqrfa (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   tau, iflag, nparts, work)
c
       double precision lftblk(1), array(1), rgtblk(1),
     *                  tau(1), work(1)
       integer nrwblk, nbloks, iflag, nparts
c     ***-----------------------------------------------------------***
c     *   This subroutine factors the ABD matrix defined in arrays    *
c     *   lftblk, array, and rgtblk using a variant of the parallel   *
c     *   SLF-QR algorithm.  On return, lftblk, array, rgtblk,        *
c     *   work, and tau contain the decomposition of the matrix       *
c     *   and factors of the elementary reflectors used to compute    *
c     *   the decompostion.  See comments in subroutine 'slf_qr'      *
```

```
c     *   for further details.                                          *
c     ***--------------------------------------------------------------***
          integer nsquar, wk1, wk2, wk3, minblk, remblk
c
          iflag = 0
c     ***--------------------------------------------------------------***
c     *      Use non-partitioned SLF-QR if requested number            *
c     *      of partitions would result in fewer than 2 blocks         *
c     *      per partition.                                            *
c     ***--------------------------------------------------------------***
          if (nbloks .lt. 2*nparts) then
             nparts = 1
          endif
c     ***--------------------------------------------------------------***
c     *      Work-space allocation:                                    *
c     *         1st/2nd-level fill-in      -   work(1)..work(wk2-1)  *
c     *         3rd-level reduced matrix   - work(wk2)..work(wk3-1)  *
c     *         temporary storage for BLAS - work(wk3)..end           *
c     *                                                               *
c     *      Total requirement:  nbloks*[(nrwblk)x(2*nrwblk)]         *
c     *                        +        [(2*nrwblk)x(2*nrwblk)]      *
c     *                        + nparts*[(2*nrwblk)x(nrwblk+1)]      *
c     ***--------------------------------------------------------------***
          nsquar = nrwblk**2
          wk1 = 1
          wk2 = wk1 + 2*nbloks*nsquar
          wk3 = wk2 + 4*nsquar
c     ***--------------------------------------------------------------***
c     *      Calculate minimum number of blocks per partition          *
c     *      Remaining blocks are distributed evenly among the         *
c     *      first partitions.                                         *
c     ***--------------------------------------------------------------***
          minblk = nbloks/nparts
          remblk = nbloks - minblk*nparts
c     ***--------------------------------------------------------------***
c     *      Three level factorization.                                *
c     ***--------------------------------------------------------------***
          call sqrf1(array,work(wk1),nrwblk,nbloks,tau,iflag,
     *              minblk,remblk,nparts,work(wk3))
          call sqrf2(array,work(wk1),nrwblk,tau,iflag,
     *              minblk,remblk,nparts,work(wk3))
          call sqrf3(lftblk,array,work(wk1),work(wk2),
     *              nrwblk,nbloks,rgtblk,tau,iflag,work(wk3))
c     ***--------------------------------------------------------------***
c     *      Set iflag to -1 if exact singularity was detected.        *
c     ***--------------------------------------------------------------***
          if (iflag .ne. 0) then
             iflag = -1
          endif
       return
       end
c-----------------------------------------------------------------------
       subroutine sqrf1 (array, fill, nrwblk, nbloks, tau, iflag,
     *                   minblk, remblk, nparts, blaws)
```

```
c
      double precision array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), tau(1),
     *                 blaws(2*nrwblk,nrwblk+1,1)
      integer nrwblk, nbloks, iflag, minblk, remblk, nparts
c     ***-----------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *                                                               *
c     *   In addition, the affix ' designates that the matrix is      *
c     *   transformed at the first level of the factorization.        *
c     ***-----------------------------------------------------------***
         integer ndoubl, kpart, kblok, base, top, info, msing
c
         ndoubl = 2*nrwblk
c     ***-----------------------------------------------------------***
c     *      Each loop 20 iteration is independent and could          *
c     *      execute concurrently with the others.                    *
c     ***-----------------------------------------------------------***
C$DOACROSS SHARE (array, fill, nrwblk, nbloks, tau, iflag,
C$&               minblk, remblk, nparts, blaws, ndoubl),
C$&        LOCAL (kpart, kblok, base, top, info)
         do 20 kpart = 1, nparts
c     ***-----------------------------------------------------------***
c     *         S_top <-- V_top                                       *
c     ***-----------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            if (kpart .gt. 1) then
               call mcopy(1,nrwblk,fill(1,1,top),blaws,
     *                    blaws,array(1,1,top-1))
            else
               call mcopy(1,nrwblk,fill(1,1,top),blaws,
     *                    blaws,array(1,1,nbloks))
            endif
c     ***-----------------------------------------------------------***
c     *      SLF-QR starts at the [top; top+1] block-row pair of      *
c     *      each partition and proceeds downward sequentially        *
c     *      to the [base-1; base] block-row pair.                    *
c     ***-----------------------------------------------------------***
            do 10 kblok = top, base-1
c     ***-----------------------------------------------------------***
c     *            [W_kblok    <-- QRfact([W_kblok                    *
c     *             V_kblok+1]'            V_kblok+1]                *
c     ***-----------------------------------------------------------***
               call DGEQRF(ndoubl,nrwblk,array(1,1,kblok),ndoubl,
     *                     tau(kblok*nrwblk+1),blaws(1,nrwblk+1,kpart),
     *                     ndoubl,info)
               iflag = msing(nrwblk,array(1,1,kblok))
c     ***-----------------------------------------------------------***
```

```
c     *            [S_kblok    <-- Q^T [S_kblok                      *
c     *             S_kblok+1]'           0    ]                     *
c     ***-----------------------------------------------------------***
                call mcopy(3,nrwblk,fill(1,1,kblok),blaws,
     *                     blaws(1,1,kpart),blaws)
                call DORMQR('L','T',ndoubl,nrwblk,nrwblk,
     *                    array(1,1,kblok),ndoubl,
     *                    tau(kblok*nrwblk+1),blaws(1,1,kpart),ndoubl,
     *                    blaws(1,nrwblk+1,kpart),ndoubl,info)
                call mcopy(4,nrwblk,fill(1,1,kblok),fill(1,1,kblok+1),
     *                     blaws(1,1,kpart),blaws)
c     ***-----------------------------------------------------------***
c     *            [T_kblok    <-- Q^T [  0                          *
c     *             W_kblok+1]'         W_kblok+1]                   *
c     ***-----------------------------------------------------------***
                call mcopy(5,nrwblk,blaws,blaws,
     *                     blaws(1,1,kpart),array(1,1,kblok+1))
                call DORMQR('L','T',ndoubl,nrwblk,nrwblk,
     *                    array(1,1,kblok),ndoubl,
     *                    tau(kblok*nrwblk+1),blaws(1,1,kpart),ndoubl,
     *                    blaws(1,nrwblk+1,kpart),ndoubl,info)
                call mcopy(6,nrwblk,fill(1,1,kblok),blaws,
     *                     blaws(1,1,kpart),array(1,1,kblok+1))
  10         continue
  20      continue
          return
          end
c-----------------------------------------------------------------------
          subroutine sqrf2 (array, fill, nrwblk, tau, iflag,
     *                      minblk, remblk, nparts, blaws)
c
          double precision array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), tau(1), blaws(2*nrwblk,1)
          integer nrwblk, iflag, minblk, remblk, nparts
c     ***-----------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *                                                               *
c     *   In addition, the affix '/'' designates that the             *
c     *   matrix was/is transformed at the first/second level         *
c     *   of the factorization.                                       *
c     ***-----------------------------------------------------------***
          integer ndoubl, kpart, base, base1, top, info, msing
c     ***-----------------------------------------------------------***
c     *      If there is only one partition, nothing needs to be      *
c     *      done at the second level of the factorization.           *
c     ***-----------------------------------------------------------***
          if (nparts .eq. 1) return
c     ***-----------------------------------------------------------***
```

```
c     *     SLF-QR starts at the [1; 2] block-row pair of the        *
c     *     second-level array and proceeds downward sequentially    *
c     *     to the [nparts-1; nparts] block-row pair.                *
c     ***------------------------------------------------------------***
          ndoubl = 2*nrwblk
          do 10 kpart = 1, nparts-1
             call partx(minblk,remblk,kpart,base,top)
             call partx(minblk,remblk,kpart+1,base1,top)
c     ***------------------------------------------------------------***
c     *            [W_base     <-- QRfact([W_base'                   *
c     *             V_base+1]''            S_base1']                 *
c     ***------------------------------------------------------------***
             call mcopy(2,nrwblk,fill(1,1,base1),blaws,
     *                  blaws,array(1,1,base))
             call DGEQRF(ndoubl,nrwblk,array(1,1,base),ndoubl,
     *               tau(base*nrwblk+1),blaws(1,nrwblk+1),ndoubl,info)
             iflag = msing(nrwblk,array(1,1,base))
c     ***------------------------------------------------------------***
c     *            [S_base    <-- Q^T [S_base'                      *
c     *             S_base1]''           0    ]                     *
c     ***------------------------------------------------------------***
             call mcopy(3,nrwblk,fill(1,1,base),blaws,
     *                  blaws,blaws)
             call DORMQR('L','T',ndoubl,nrwblk,nrwblk,
     *                   array(1,1,base),ndoubl,tau(base*nrwblk+1),
     *                   blaws,ndoubl,blaws(1,nrwblk+1),ndoubl,info)
             call mcopy(4,nrwblk,fill(1,1,base),fill(1,1,base1),
     *                  blaws,blaws)
c     ***------------------------------------------------------------***
c     *            [T_base    <-- Q^T [  0                          *
c     *             W_base1]''         W_base1']                    *
c     ***------------------------------------------------------------***
             call mcopy(5,nrwblk,blaws,blaws,
     *                  blaws,array(1,1,base1))
             call DORMQR('L','T',ndoubl,nrwblk,nrwblk,
     *                   array(1,1,base),ndoubl,tau(base*nrwblk+1),
     *                   blaws,ndoubl,blaws(1,nrwblk+1),ndoubl,info)
             call mcopy(6,nrwblk,fill(1,1,base),blaws,
     *                  blaws,array(1,1,base1))
  10     continue
         return
         end
c-----------------------------------------------------------------------
         subroutine sqrf3 (lftblk, array, fill, rdcmx,
     *                     nrwblk, nbloks, rgtblk, tau, iflag, blaws)
c
         double precision lftblk(nrwblk,1), array(2*nrwblk,nrwblk,1),
     *                    fill(nrwblk,2*nrwblk,1), rdcmx(2*nrwblk,1),
     *                    rgtblk(nrwblk,1), tau(1), blaws(2*nrwblk,1)
         integer nrwblk, nbloks, iflag
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:            *
c     *                                                              *
c     *      B_a, B_b <=> lftblk, rgtblk                             *
```

```
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)            *
c     *                                                               *
c     *   In addition, the affix ''/''' designates that the           *
c     *   matrix was/is transformed at the second/third level         *
c     *   of the factorization.                                       *
c     ***-------------------------------------------------------------***
          integer i, j, ndoubl, info
c     ***-------------------------------------------------------------***
c     *      The third-level block-array is of the form               *
c     *                                                               *
c     *            [   B_a        B_b                                *
c     *             S_nbloks'' W_nbloks'']                            *
c     *                                                               *
c     *      The QR-factorization of this block-array is stored       *
c     *      in the rdcmx(,) work-space.  The scalar factors of the   *
c     *      elementary reflectors are stored in [tau(1..nrwblk);     *
c     *      tau(nbloks*nrwblk+1..(nbloks+1)*nrwblk)].                *
c     ***-------------------------------------------------------------***
          ndoubl = 2*nrwblk
          do 20 j = 1, nrwblk
             do 10 i = 1, nrwblk
                rdcmx(i,j) = lftblk(i,j)
                rdcmx(i,nrwblk+j) = rgtblk(i,j)
                rdcmx(nrwblk+i,j) = fill(i,j,nbloks)
                rdcmx(nrwblk+i,nrwblk+j) = array(i,j,nbloks)
  10         continue
  20      continue
          call DGEQRF(ndoubl,ndoubl,rdcmx,ndoubl,blaws,
     *               blaws(1,nrwblk+1),ndoubl,info)
          call DCOPY(nrwblk,blaws,1,tau,1)
          call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                      tau(nbloks*nrwblk+1),1)
c     ***-------------------------------------------------------------***
c     *      Check for exact singularity.                             *
c     ***-------------------------------------------------------------***
          do 30 i = 1, ndoubl
             if (rdcmx(i,i) .eq. 0.d0) iflag = i
  30      continue
       return
       end
c-----------------------------------------------------------------------
       subroutine sqrsl (lftblk, array, nrwblk, nbloks, rgtblk,
     *                   b, tau, nparts, work)
c
       double precision lftblk(1), array(1), rgtblk(1),
     *                   b(1), tau(1), work(1)
       integer nrwblk, nbloks, nparts
c     ***-------------------------------------------------------------***
c     *   Given the factors of ABD matrix A computed by subroutine    *
c     *   'sqrfa' and stored in arrays lftblk, array, rgtblk,         *
```

```
c     *   work and tau, this subroutine solves the linear system     *
c     *   A x = b.  b is overwritten with x.  See comments in        *
c     *   subroutine 'slf_qr' for further details.                   *
c     ***------------------------------------------------------------***
          integer nsquar, wk1, wk2, wk3, minblk, remblk
c     ***------------------------------------------------------------***
c     *      Work-space allocation:                                  *
c     *         1st/2nd-level fill-in      -   work(1)..work(wk2-1)  *
c     *         3rd-level reduced matrix   - work(wk2)..work(wk3-1)  *
c     *         temporary storage for BLAS - work(wk3)..end          *
c     *                                                             *
c     *      Total requirement:  nbloks*[(nrwblk)x(2*nrwblk)]       *
c     *                        +        [(2*nrwblk)x(2*nrwblk)]     *
c     *                        + nparts*[(2*nrwblk)x(nrwblk+1)]     *
c     ***------------------------------------------------------------***
          nsquar = nrwblk**2
          wk1 = 1
          wk2 = wk1 + 2*nbloks*nsquar
          wk3 = wk2 + 4*nsquar
c     ***------------------------------------------------------------***
c     *      Calculate minimum number of blocks per partition       *
c     *      Remaining blocks are distributed evenly among the      *
c     *      first partitions.                                      *
c     ***------------------------------------------------------------***
          minblk = nbloks/nparts
          remblk = nbloks - minblk*nparts
c     ***------------------------------------------------------------***
c     *      Three level forward elimination.                       *
c     ***------------------------------------------------------------***
          call sqrsf1(array,nrwblk,b,tau,
     *               minblk,remblk,nparts,work(wk3))
          call sqrsf2(array,nrwblk,b,tau,
     *               minblk,remblk,nparts,work(wk3))
          call sqrsf3(work(wk2),nrwblk,nbloks,b,b,tau,work(wk3))
c     ***------------------------------------------------------------***
c     *      Three level back-solve.                                *
c     ***------------------------------------------------------------***
          call sqrsb3(work(wk2),nrwblk,nbloks,b,b,work(wk3))
          call sqrsb2(array,work(wk1),nrwblk,nbloks,b,
     *               minblk,remblk,nparts,work(wk3))
          call sqrsb1(array,work(wk1),nrwblk,b,
     *               minblk,remblk,nparts,work(wk3))
      return
      end
c-----------------------------------------------------------------------
      subroutine sqrsf1 (array, nrwblk, phi, tau,
     *                   minblk, remblk, nparts, blaws)
c
      double precision array(2*nrwblk,nrwblk,1), phi(1),
     *                 tau(1), blaws(2*nrwblk,nrwblk+1,1)
      integer nrwblk, minblk, remblk, nparts
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:           *
c     *                                                             *
```

```
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1              *
c     *       V_k+1]                                                  *
c     *          phi_k <=> phi(k*nrwblk+1)                            *
c     *                                                               *
c     *   In addition, the affix ' designates that the vector is      *
c     *   transformed at the first level of the forward solve.        *
c     ***-------------------------------------------------------------***
          integer ndoubl, kpart, kblok, base, top, info
c
          ndoubl = 2*nrwblk
c     ***-------------------------------------------------------------***
c     *      Each loop 20 iteration is independent and could          *
c     *      execute concurrently with the others.                    *
c     ***-------------------------------------------------------------***
C$DOACROSS SHARE (array, nrwblk, phi, tau,
C$&               minblk, remblk, nparts, blaws, ndoubl),
C$&        LOCAL (kpart, kblok, base, top, info)
          do 20 kpart = 1, nparts
c     ***-------------------------------------------------------------***
c     *      Forward elimination starts at [phi_top; phi_top+1]       *
c     *      of each partition and proceeds downward sequentially     *
c     *      to [phi_base-1; phi_base].                               *
c     ***-------------------------------------------------------------***
          call partx(minblk,remblk,kpart,base,top)
          do 10 kblok = top, base-1
c     ***-------------------------------------------------------------***
c     *           [phi_kblok    <-- Q^T [phi_kblok                   *
c     *            phi_kblok+1]'         phi_kblok+1]                *
c     *                                                               *
c     *      Q^T is resurrected from nrwblk elementary reflectors     *
c     *      stored in [W_kblok; V_kblok+1] and tau_kblok.            *
c     ***-------------------------------------------------------------***
             call DORMQR('L','T',ndoubl,1,nrwblk,
     *           array(1,1,kblok),ndoubl,tau(kblok*nrwblk+1),
     *           phi(kblok*nrwblk+1),ndoubl,
     *           blaws(1,nrwblk+1,kpart),ndoubl,info)
  10      continue
  20    continue
        return
        end
c-----------------------------------------------------------------------
        subroutine sqrsf2 (array, nrwblk, phi, tau,
     *                    minblk, remblk, nparts, blaws)
c
        double precision array(2*nrwblk,nrwblk,1), phi(1),
     *                  tau(1), blaws(2*nrwblk,1)
        integer nrwblk, minblk, remblk, nparts
c     ***-------------------------------------------------------------***
c     *   The following notation is used in the comments:             *
c     *                                                               *
c     *      [W_nbloks <=> array(1,1,nbloks)                          *
c     *       V_1]                                                    *
```

```
c    *      [W_k      <=> array(1,1,k), k = 1..nbloks-1             *
c    *       V_k+1]                                               *
c    *          phi_k <=> phi(k*nrwblk+1)                         *
c    *                                                            *
c    *   In addition, the affix '//'' designates that the         *
c    *   vector was/is transformed at the first/second level      *
c    *   of the forward solve.                                    *
c    ***------------------------------------------------------------***
         integer ndoubl, kpart, base, base1, top, info
c    ***------------------------------------------------------------***
c    *      If there is only one partition, nothing needs to be   *
c    *      done at the second level of the forward solve.        *
c    ***------------------------------------------------------------***
         if (nparts .eq. 1) return
c    ***------------------------------------------------------------***
c    *    Forward elimination starts at [phi_base_1; phi_base_2]  *
c    *    of the second-level system and proceeds downward        *
c    *    sequentially to [phi_base_nparts-1; phi_base_nparts].   *
c    ***------------------------------------------------------------***
         ndoubl = 2*nrwblk
         do 10 kpart = 1, nparts-1
            call partx(minblk,remblk,kpart,base,top)
            call partx(minblk,remblk,kpart+1,base1,top)
c    ***------------------------------------------------------------***
c    *          [phi_base    <-- Q^T [phi_base''                  *
c    *           phi_base1]''         phi_base1']                 *
c    *                                                            *
c    *      Q^T is resurrected from nrwblk elementary reflectors  *
c    *      stored in [W_base; V_base+1] and tau_base.            *
c    ***------------------------------------------------------------***
            call DCOPY(nrwblk,phi(base*nrwblk+1),1,blaws,1)
            call DCOPY(nrwblk,phi(base1*nrwblk+1),1,
     *                        blaws(nrwblk+1,1),1)
            call DORMQR('L','T',ndoubl,1,nrwblk,
     *                  array(1,1,base),ndoubl,tau(base*nrwblk+1),
     *                  blaws,ndoubl,blaws(1,nrwblk+1),ndoubl,info)
            call DCOPY(nrwblk,blaws,1,phi(base*nrwblk+1),1)
            call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                        phi(base1*nrwblk+1),1)
   10    continue
         return
         end
c-----------------------------------------------------------------------
         subroutine sqrsf3 (rdcmx, nrwblk, nbloks, beta, phi, tau, blaws)
c
         double precision rdcmx(2*nrwblk,1), beta(1), phi(1),
     *                  tau(1), blaws(2*nrwblk,1)
         integer nrwblk, nbloks
c    ***------------------------------------------------------------***
c    *   The following notation is used in the comments:          *
c    *                                                            *
c    *          beta <=> beta(1) (<=> phi_0)                      *
c    *         phi_k <=> phi(k*nrwblk+1), k >= 1                  *
c    *                                                            *
```

```
c     *   In addition, the affix ''/''' designates that the          *
c     *   vector was/is transformed at the second/third level        *
c     *   of the forward solve.                                      *
c     ***------------------------------------------------------------***
          integer ndoubl, info
c     ***------------------------------------------------------------***
c     *     The right-hand side of the third-level system is         *
c     *     multiplied through by the transpose of the orthogonal    *
c     *     factor of the third-level block-array:                   *
c     *                                                              *
c     *        [beta          <-- Qˆ T [beta                         *
c     *         phi_nbloks]'''         phi_nbloks'']                 *
c     *                                                              *
c     *     Qˆ T is resurrected from 2*nrwblk elementary reflectors   *
c     *     stored in rdcmx(,) and [tau_0; tau_nbloks].              *
c     ***------------------------------------------------------------***
          ndoubl = 2*nrwblk
          call DCOPY(nrwblk,tau,1,blaws(1,2),1)
          call DCOPY(nrwblk,tau(nbloks*nrwblk+1),1,
     *                      blaws(nrwblk+1,2),1)
          call DCOPY(nrwblk,beta,1,blaws,1)
          call DCOPY(nrwblk,phi(nbloks*nrwblk+1),1,
     *                      blaws(nrwblk+1,1),1)
          call DORMQR('L','T',ndoubl,1,ndoubl,
     *               rdcmx,ndoubl,blaws(1,2),blaws,ndoubl,
     *               blaws(1,nrwblk+1),ndoubl,info)
          call DCOPY(nrwblk,blaws,1,beta,1)
          call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                      phi(nbloks*nrwblk+1),1)
          return
          end
c-----------------------------------------------------------------------
          subroutine sqrsb3 (rdcmx, nrwblk, nbloks, beta, phi, blaws)
c
          double precision rdcmx(2*nrwblk,1), beta(1),
     *                 phi(1), blaws(2*nrwblk,1)
          integer nrwblk, nbloks
c     ***------------------------------------------------------------***
c     *   The following notation is used in the comments:            *
c     *                                                              *
c     *          beta <=> beta(1) (<=> phi_0)                        *
c     *         phi_k <=> phi(k*nrwblk+1), k >= 1                    *
c     *                                                              *
c     *   Since [beta; phi] is overwritten with the solution,        *
c     *                                                              *
c     *      y_a <=> y_0      <=> beta                               *
c     *              y_k      <=> phi_k,  k = 1, nbloks-1            *
c     *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)               *
c     *                                                              *
c     *   In addition, the affix ''' designates that the solution    *
c     *   vector is obtained at the third level of the back-solve.   *
c     ***------------------------------------------------------------***
          integer ndoubl
c     ***------------------------------------------------------------***
```

```
c     *      y_a''' and y_b''' are obtained by solving the upper-    *
c     *      triangular third-level system                           *
c     *                                                              *
c     *         rdcmx(,) [y_a  = [beta                               *
c     *                   y_b]    phi_nbloks]                        *
c     ***----------------------------------------------------------***
          ndoubl = 2*nrwblk
          call DCOPY(nrwblk,beta,1,blaws,1)
          call DCOPY(nrwblk,phi(nbloks*nrwblk+1),1,
     *                     blaws(nrwblk+1,1),1)
          call DTRSM('L','U','N','N',ndoubl,1,1.d0,
     *               rdcmx,ndoubl,blaws,ndoubl)
          call DCOPY(nrwblk,blaws,1,beta,1)
          call DCOPY(nrwblk,blaws(nrwblk+1,1),1,
     *                     phi(nbloks*nrwblk+1),1)
       return
       end
c-----------------------------------------------------------------------
       subroutine sqrsb2 (array, fill, nrwblk, nbloks, phi,
     *                    minblk, remblk, nparts, blaws)
c
       double precision array(2*nrwblk,nrwblk,1),
     *              fill(nrwblk,2*nrwblk,1), phi(1), blaws(2*nrwblk,1)
       integer nrwblk, nbloks, minblk, remblk, nparts
c     ***----------------------------------------------------------***
c     *   The following notation is used in the comments:            *
c     *                                                              *
c     *      [W_nbloks <=> array(1,1,nbloks)                         *
c     *       V_1]                                                   *
c     *      [W_k      <=> array(1,1,k), k = 1..nbloks-1             *
c     *       V_k+1]                                                 *
c     *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)           *
c     *          phi_k <=> phi(k*nrwblk+1)                           *
c     *                                                              *
c     *   Since phi is overwritten with the solution,                *
c     *                                                              *
c     *      y_a <=> y_0      <=> phi_0  (<=> beta in sqrsb3.f)      *
c     *              y_k      <=> phi_k,  k = 1, nbloks-1            *
c     *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)               *
c     *                                                              *
c     *   In addition, the affix '''/''' designates that the         *
c     *   solution vector was/is obtained at the third/second        *
c     *   level of the back-solve.                                   *
c     ***----------------------------------------------------------***
          integer ndoubl, kpart, base, basep, top
c     ***----------------------------------------------------------***
c     *      If there is only one partition, nothing needs to be     *
c     *      done at the second level of the back-solve.             *
c     ***----------------------------------------------------------***
          if (nparts .eq. 1) return
c     ***----------------------------------------------------------***
c     *      Back-solve starts at the second-last block-row of the   *
c     *      second-level system and proceeds upward sequentially    *
c     *      to the first block-row.                                 *
```

```
c    ***-----------------------------------------------------------***
         ndoubl = 2*nrwblk
         base = nbloks
         call DCOPY(nrwblk,phi,1,blaws,1)
         do 10 kpart = nparts-1, 1, -1
            basep = base
            call partx(minblk,remblk,kpart,base,top)
c    ***-----------------------------------------------------------***
c    *   phi_base <-- phi_base - S_base y_0''' - T_base y_basep''   *
c    ***-----------------------------------------------------------***
            call DCOPY(nrwblk,phi(basep*nrwblk+1),1,
     *                       blaws(nrwblk+1,1),1)
            call DGEMV('N',nrwblk,ndoubl,
     *                 -1.d0,fill(1,1,base),nrwblk,
     *                 blaws,1,1.d0,phi(base*nrwblk+1),1)
c    ***-----------------------------------------------------------***
c    *   y_base'' is now obtained by solving the upper-triangular   *
c    *   nrwblkxnrwblk system  W_base y_base = phi_base             *
c    ***-----------------------------------------------------------***
            call DTRSM('L','U','N','N',nrwblk,1,1.d0,
     *                 array(1,1,base),ndoubl,
     *                 phi(base*nrwblk+1),nrwblk)
  10     continue
         return
         end
c-----------------------------------------------------------------------
         subroutine sqrsb1 (array, fill, nrwblk, phi,
     *                   minblk, remblk, nparts, blaws)
c
         double precision array(2*nrwblk,nrwblk,1),
     *                 fill(nrwblk,2*nrwblk,1), phi(1),
     *                 blaws(2*nrwblk,nrwblk+1,1)
         integer nrwblk, minblk, remblk, nparts
c    ***-----------------------------------------------------------***
c    *   The following notation is used in the comments:            *
c    *                                                              *
c    *      [W_nbloks <=> array(1,1,nbloks)                         *
c    *       V_1]                                                   *
c    *      [W_k      <=> array(1,1,k), k = 1..nbloks-1             *
c    *       V_k+1]                                                 *
c    *      S_k, T_k  <=> fill(1,1,k), fill(1,nrwblk+1,k)           *
c    *          phi_k <=> phi(k*nrwblk+1)                           *
c    *                                                              *
c    *   Since phi is overwritten with the solution,                *
c    *                                                              *
c    *      y_a <=> y_0      <=> phi_0  (<=> beta in sqrsb3.f)      *
c    *              y_k      <=> phi_k,  k = 1, nbloks-1            *
c    *      y_b <=> y_nbloks <=> phi(nbloks*nrwblk+1)               *
c    *                                                              *
c    *   In addition, the affix ''/'' designates that the           *
c    *   solution vector was/is obtained at the second/first        *
c    *   level of the back-solve.                                   *
c    ***-----------------------------------------------------------***
         integer ndoubl, kpart, kblok, base, top
```

```
c
         ndoubl = 2*nrwblk
c    ***---------------------------------------------------------***
c    *      Each loop 20 iteration is independent and could         *
c    *      execute concurrently with the others.                   *
c    ***---------------------------------------------------------***
C$DOACROSS SHARE (array, fill, nrwblk, phi,
C$&              minblk, remblk, nparts, blaws, ndoubl),
C$&       LOCAL (kpart, kblok, base, top)
         do 20 kpart = 1, nparts
c    ***---------------------------------------------------------***
c    *      Back-solve starts at the second-last block-row of       *
c    *      each partition and proceeds upward sequentially to      *
c    *      the first block-row.                                    *
c    ***---------------------------------------------------------***
            call partx(minblk,remblk,kpart,base,top)
            call DCOPY(nrwblk,phi((top-1)*nrwblk+1),1,
     *                        blaws(1,1,kpart),1)
            do 10 kblok = base-1, top, -1
c    ***---------------------------------------------------------***
c    *         phi_kblok <-- phi_kblok - S_kblok y_top-1''          *
c    *                                - T_kblok y_kblok+1'         *
c    ***---------------------------------------------------------***
               call DCOPY(nrwblk,phi((kblok+1)*nrwblk+1),1,
     *                           blaws(nrwblk+1,1,kpart),1)
               call DGEMV('N',nrwblk,ndoubl,
     *                    -1.d0,fill(1,1,kblok),nrwblk,
     *                    blaws(1,1,kpart),1,1.d0,
     *                    phi(kblok*nrwblk+1),1)
c    ***---------------------------------------------------------***
c    *   y_kblok' is now obtained by solving the upper-triangular   *
c    *   nrwblkxnrwblk system  W_kblok y_kblok = phi_kblok          *
c    ***---------------------------------------------------------***
               call DTRSM('L','U','N','N',nrwblk,1,1.d0,
     *                    array(1,1,kblok),ndoubl,
     *                    phi(kblok*nrwblk+1),nrwblk)
  10        continue
  20     continue
      return
      end
c-----------------------------------------------------------------------
      subroutine mcopy (sel, n, C, D, E, F)
c
      double precision C(n,1), D(n,1), E(2*n,1), F(2*n,1)
      integer sel, n
c    ***---------------------------------------------------------***
c    *   'mcopy' performs the matrix copy selected by sel.          *
c    *   Note:  Due to the nature of the storage organization,      *
c    *   vectorized copying (Lapack's DCOPY) is not possible.       *
c    *                                                              *
c    *    on entry                                                  *
c    *                                                              *
c    *         sel [integer]                                        *
c    *             Copy selection (see below for details).          *
```

```
c    *                                                               *
c    *           n [integer]                                         *
c    *             The number of rows in C and D and the number      *
c    *             of columns in E and F.  There are 2*n columns     *
c    *             in C and D and 2*n rows in E and F.               *
c    *                                                               *
c    *  C, D, E, F [double precision                                 *
c    *              (n,2*n), (n,2*n), (2*n,n), (2*n,n)]              *
c    *             The matrices before copying.                      *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *         sel [integer]                                         *
c    *             Unchanged.                                        *
c    *                                                               *
c    *           n [integer]                                         *
c    *             Unchanged.                                        *
c    *                                                               *
c    *  C, D, E, F [double precision                                 *
c    *              (n,2*n), (n,2*n), (2*n,n), (2*n,n)]              *
c    *             The matrices after copying.                       *
c    ***---------------------------------------------------------***
         integer i, j, j1
c    ***---------------------------------------------------------***
c    *      The following notation is used in the comments:          *
c    *                                                               *
c    *             C <=> [C1 C2]    D <=> [D1 D2]                   *
c    *             E <=> [E1        F <=> [F1                       *
c    *                    E2]              F2]                     *
c    *                                                               *
c    *      Ck, Dk, Ek and Fk, k = 1, 2, are each nxn matrices.      *
c    *      (0) is the nxn zero matrix.                              *
c    ***---------------------------------------------------------***
         go to (100, 200, 300, 400, 500, 600) sel
c    ***---------------------------------------------------------***
c    *         Copy selection #1:  C1 <- F2                          *
c    ***---------------------------------------------------------***
 100         do 120 j = 1, n
                do 110 i = 1, n
                   C(i,j) = F(n+i,j)
 110            continue
 120         continue
         return
c    ***---------------------------------------------------------***
c    *         Copy selection #2:  F2 <- C1                          *
c    ***---------------------------------------------------------***
 200         do 220 j = 1, n
                do 210 i = 1, n
                   F(n+i,j) = C(i,j)
 210            continue
 220         continue
         return
c    ***---------------------------------------------------------***
c    *         Copy selection #3:  E1 <- C1                          *
```

```
c    *                            E2 <- (0)                          *
c    ***-----------------------------------------------------------***
 300        do 320 j = 1, n
               do 310 i = 1, n
                  E(i,j) = C(i,j)
                  E(n+i,j) = 0.d0
 310           continue
 320        continue
         return
c    ***-----------------------------------------------------------***
c    *         Copy selection #4:  C1 <- E1                         *
c    *                             D1 <- E2                         *
c    ***-----------------------------------------------------------***
 400        do 420 j = 1, n
               do 410 i = 1, n
                  C(i,j) = E(i,j)
                  D(i,j) = E(n+i,j)
 410           continue
 420        continue
         return
c    ***-----------------------------------------------------------***
c    *         Copy selection #5:  E1 <- (0)                        *
c    *                             E2 <- F1                         *
c    ***-----------------------------------------------------------***
 500        do 520 j = 1, n
               do 510 i = 1, n
                  E(i,j) = 0.d0
                  E(n+i,j) = F(i,j)
 510           continue
 520        continue
         return
c    ***-----------------------------------------------------------***
c    *         Copy selection #6:  C2 <- E1                         *
c    *                             F1 <- E2                         *
c    ***-----------------------------------------------------------***
 600        do 620 j = 1, n
               j1 = n + j
               do 610 i = 1, n
                  C(i,j1) = E(i,j)
                  F(i,j) = E(n+i,j)
 610           continue
 620        continue
         return
       end
c-----------------------------------------------------------------------
       integer function msing (n, A)
c
       double precision A(1)
       integer n
c    ***-----------------------------------------------------------***
c    *   'msing' checks for exact singularity in the nxn upper-     *
c    *   triangle of 2*nxn matrix A.                                *
c    *                                                              *
c    *    on entry                                                  *
```

```
c    *                                                               *
c    *           n [integer]                                         *
c    *             The number of columns and half the number of      *
c    *             rows in A.  This is implicit -- A is accessed     *
c    *             as a 1D array inside this function.               *
c    *                                                               *
c    *           A [double precision(2*n**2)]                        *
c    *             Only the diagonal of the nxn upper-triangle       *
c    *             is accessed.                                      *
c    *                                                               *
c    *   on return                                                   *
c    *                                                               *
c    *           n [integer]                                         *
c    *             Unchanged.                                        *
c    *                                                               *
c    *           A [double precision(2*n**2)]                        *
c    *             Unchanged.                                        *
c    *                                                               *
c    *        msing [integer]                                        *
c    *              = k if the k-th element on the diagonal of       *
c    *                the nxn upper-triangle of A is exactly zero    *
c    *                (note:  there may be other zeros),             *
c    *              = 0 otherwise.                                   *
c    ***---------------------------------------------------------------***
         integer ndoubl, i, k
c
         ndoubl = 2*n
         do 10 k = 1, n
            i = (k-1)*ndoubl + k
            if (A(i) .eq. 0.d0) then
               msing = k
               return
            end if
   10    continue
         msing = 0
      return
      end
c-----------------------------------------------------------------------
      subroutine partx (minblk, remblk, k, base, top)
c
      integer minblk, remblk, k, base, top
c    ***---------------------------------------------------------------***
c    *   'partx' calculates the index of the base and top block      *
c    *   of the k-th partition.  The indexing scheme assumes that    *
c    *   nbloks >= 2*nparts, so minblk >= 2 and remblk >= 0.         *
c    *                                                               *
c    *    on entry                                                   *
c    *                                                               *
c    *      minblk [integer]                                         *
c    *             minimum number of blocks/partition                *
c    *             (minblk = nbloks/nparts)                          *
c    *                                                               *
c    *      remblk [integer]                                         *
c    *             first remblk partitions have minblk+1 blocks      *
```

```
c    *             (remblk = nbloks - minblk*nparts)                *
c    *                                                              *
c    *           k [integer]                                        *
c    *                                                              *
c    *   on return                                                  *
c    *                                                              *
c    *        base [integer]                                        *
c    *             index of base block of k-th partition            *
c    *                                                              *
c    *         top [integer]                                        *
c    *             index of top block of k-th partition             *
c    ***------------------------------------------------------------***
         integer khigh, klow
c
         if (k .le. remblk) then
            khigh = k
            klow = 0
         else
            khigh = remblk
            klow = k - khigh
         endif
         base = khigh*(minblk+1) + klow*minblk
         top = base - minblk + 1
         if (k .le. remblk) then
            top = top - 1
         endif
       return
       end
```

Bibliography

[Amod 00] P. Amodio, J.R. Cash, G. Roussos, R.W. Wright, G. Fairweather, I. Gladwell, G.L. Kraut, M. Paprzycki, *Almost Block Diagonal linear systems: sequential and parallel solution techniques, and applications*, Numer. Linear Algebra Appl., 7 (2000), pp. 275-317.

[Asch 81] U.M. Ascher, J. Christiansen, R.D. Russell, *Collocation software for boundary-value ODEs*, ACM Trans. Math. Software, 7 (1981), pp. 209-222.

[Asch 88] U.M. Ascher, R.M.M. Mattheij, R.D. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, 1988.

[Asch 91] U.M. Ascher, S.Y.P. Chan, *On parallel methods for boundary value ODEs*, Computing 46/1 (1991), pp. 1-17.

[Bade 87] G. Bader, U.M. Ascher, *A new basis implementation for a mixed order boundary value ODE solver*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 483-500.

[Benn 90] K.R. Bennett, G. Fairweather, *PCOLNEW: A parallel boundary-value ODE code for shared-memory machines*, Univ. of Kentucky, Center for Computational Sciences, T.R. CCS-90-8, 1990.

[deBo 80] C. de Boor, R. Weiss, *SOLVEBLOK: A package for solving almost block diagonal linear systems*, ACM Trans. Math. Software, 6 (1980), pp. 80-87.

[Diaz 83] J.C. Diaz, G. Fairweather, P. Keast, *Algorithm 603. COLROW and ARCECO: FORTRAN packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination*, ACM Trans. Math. Software 9/3 (1983), pp. 376-380.

[Enri 96] W.H. Enright, P.H. Muir, *Runge-Kutta software with defect control for boundary value ODEs*, SIAM J. Sci. Comput. 17/2 (1996), pp. 479-497.

[Gear 88] C.W. Gear, *Massive parallelism across the method in ODEs*, Univ. of Illinois, Computer Science Dept., T.R. UIUCDCS-R-88-1442, 1988.

[Golu 83] G. Golub, C. van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1983.

[Hell 76] D. Heller, *Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems*, SIAM J. Numer. Anal., 13 (1976), pp. 484-496.

[Hock 65] R. Hockney, *A Fast Direct Solution of Poisson's Equation Using Fourier Analysis*, J. ACM, 12 (1965), pp. 95-113.

[Hock 70] R. Hockney, *The Potential Calculation and Some Applications*, Meth. Comput. Phys., 9 (1970), pp. 135-211.

[Lent 77] M. Lentini, V. Pereyra, *An adaptive finite difference solver for nonlinear two-point boundary value problems with mild boundary layers*, SIAM J. Numer. Anal., 14 (1977), pp. 91-111.

[Lent 89] M. Lentini, *Parallel solution of special large block tridiagonal systems: TPBVP*, manuscript (1989).

[Matt 85] R.M.M. Mattheij, *Decoupling and stability of algorithms for boundary value problems*, SIAM Review, 27 (1985), pp. 1-44.

[Muir 91] P.H. Muir, *Private communication.*

[Muir 03] P.H. Muir, R.N. Pancer, K.R. Jackson, *PMIRKDC: a parallel mono-implicit Runge-Kutta code with defect control for Boundary Value ODEs*, Parallel Computing, 29 (2003), pp. 711-741.

[Panc 92] R.N. Pancer, K.R. Jackson, *The parallel solution of ABD systems arising in numerical methods for BVPs for ODEs*, Univ. of Toronto, Dept. of Computer Science, T.R. 255/91, 1992.

[Papr 91] M. Paprzycki, I. Gladwell, *Solving almost block diagonal systems on parallel computers*, Parallel Comput. 17/2 (1991), pp. 133-153.

[Sche 84] U. Schendel, *Introduction to Numerical Methods for Parallel Computers*, Ellis Horwood, New York, 1984.

[Wrig 90] S.J. Wright, V. Pereyra, *Adaptation of a two-point boundary value problem solver to a vector-multiprocessor environment*, SIAM J. Sci. Stat. Comput. 11/3 (1990), pp. 425-449.

[Wrig 92] S.J. Wright, *Stable parallel algorithms for two-point boundary value problems*, SIAM J. Sci. Stat. Comput. 13/3 (1992), pp. 742-764.

[Wrig 93] S.J. Wright, *A collection of problems for which Gaussian elimination with partial pivoting is unstable*, SIAM J. Sci. Comput. 14/1 (1993), pp. 231-238.

[Wrig 94] S.J. Wright, *Stable parallel elimination for boundary value ODEs*, Numer. Math. 67/4 (1994), pp. 521-535.