PARALLEL COMPUTATION OF PARTIAL DIFFERENTIAL EQUATIONS ON DISTRIBUTED MEMORY MACHINES

Christina C. Christara
Department of Computer Sciences
University of Toronto
Toronto, Ontario, Canada, M5S 1A4
e-mail: ccc@cs.toronto.edu

Abstract

We review a number of parallel methods for the solution of elliptic Boundary Value Problems (BVPs). The methods considered are of coarse or medium grain parallelism and fixed grid discretisation. The cases where the degree of parallelism grows linearly with the size of the problem in all its dimensions or in fewer dimensions are considered. We also consider the parallelism in both the discretisation of the BVP phase and the solution of the resulting linear system phase. We discuss the assignment of data to processors and the communication requirements of the methods, when implemented on distributed memory machines. Based on a general model for a distributed memory machine, we develop a framework in order to classify the communication requirements of the methods considered. We study the communication complexity of the methods on a general distributed machine model, and present performance results on the iPSC/2 hypercube.

1. Introduction.

Linear elliptic Boundary Value Problems (BVPs) are described by an elliptic Partial Differential Equation (PDE) defined in some domain Ω

$$\mathbb{L}u(x) = g(x) \quad \text{in} \quad \Omega \tag{1.1}$$

and some boundary conditions defined on the boundary $\partial\Omega$ of Ω

$$\mathbb{B}u(x) = \gamma(x) \quad \text{on} \quad \partial\Omega \tag{1.2}$$

where $\mathbb{L}$ is a linear elliptic partial differential operator, $\mathbb{B}$ is a boundary differential operator, g and γ are given functions of x (multi-dimensional) and u is the unknown function of x.

The numerical solution of BVPs can be viewed as a three-phase processing. At first the continuous problem (1.1)-(1.2) is discretised over a set of discretisation points or elements of $\overline{\Omega} \equiv \Omega \cup \partial\Omega$, using some discretisation method (finite difference, finite element, etc). This is called the *discretisation* phase. The result is a set of (discrete) linear equations. The number of equations depends on how fine the discretisation is, but the number of unknowns per equation is often small and independent of the number of equations. In the second phase, called *indexing*, an ordering of the equations and unknowns is adopted, thus giving to the set of equations the form of a system, with a particular matrix sparsity pattern. In most cases the ordering of the equations and unknowns reflects the ordering of the points or elements of the discretisation. The third phase, referred to as *solution*, involves the solution of the sparse system of equations. This is the asymptotically most time consuming phase, while the discretisation phase comes second in computational effort.

Many discretisation methods are expressed graphically through stencils, i.e. a set of (neighbouring) points appropriately positioned in the (multidimensional) space to which Ω belongs. The stencil points correspond to equations and/or unknowns of the system, and are connected by lines, showing the dependency between unknowns of the system. The number of points in the stencil of a particular discretisation method shows the number of nonzero entries in the left side of each linear equation obtained by applying the method to the BVP. When appropriate coefficients are associated with every stencil point, we get *computational molecules* [Ames77]. Figure 1.1(a, b, c) shows the (left side) stencils of some common BVP discretisation methods. Note that such stencils can be constructed for the right side of the equations too. The most common right side stencil consists of one point, i.e. each component of the right side is computed by appropriate function evaluations on one point only. There exist though discretisation methods that require function evaluations on several points, thus building a stencil of dimension higher than zero for the right side of the system. Figure 1.1(d, e, f) shows the right side stencils of some common BVP discretisation methods. A complete description of a discretisation method can be obtained by its right and left side computational molecules. Naturally, both the left and right side stencils of Figure 1.1 are for a general interior of Ω equation. Different stencils might apply on or near the boundary.

Figure 1.1. Some left side dependency stencils: (a) 5-point-star finite difference, (b) 9-point-star, HODIE [Lync80], quadratic spline collocation, two-step quadratic spline collocation [Chri90a], linear spline Galerkin. (c) quadratic spline Galerkin, one-step quadratic spline collocation [Chri90a].
Some right side dependency stencils: (d) most common methods, (e) two-step quadratic spline collocation [Chri90a], HODIE 4-th order [Lync80], (f) HODIE 6-th order [Lync80].

There are several ways to develop and/or implement algorithms for solving a given problem in parallel. Many concentrate on the specific physical properties of the problem, or of a serial algorithm which solves it, some study the computation and dependency graph of a serial algorithm, which solves the problem, others let a compiler detect the parallelism in a given serial code, while some are driven by general concepts or language features (e.g. abstractions, tuple-space, etc.).

In the case of linear elliptic BVPs, we find it more convenient to think of the physical properties of the problem or of an algorithm which solves it, for example of the geometry of the domain of problem definition, the discretisation points or elements, the discretisation stencils, the matrix elements, rows, columns, blocks, etc. Based on this, we classify parallel methods for BVPs in the following two categories. The *geometry decomposition* methods, which split the computation involved in solving the BVP by decomposing the domain of problem definition either into overlapping (*Schwarz splitting*) or non-overlapping (*domain decomposition, Schur complement*) subdomains, and the *matrix partitioning* methods, which exploit the parallelism involved in solving the BVP by studying the dependency between unknowns (or discretisation points, matrix elements, rows and columns) of the linear system of the solution phase, either with natural ordering or with other orderings (e.g. *multicolour* ordering). The methods of both categories, although motivated differently, result in the partitioning of both the

geometric constructs of the problem as well as the algorithmic ones.

Distributed memory machines are quite widely used today for providing parallel capabilities at reasonable cost. The main source of speedup degradation in some implementation of a parallel algorithm on a distributed memory machine is usually the time spent in communication between processors, that is, the time required for an amount of data to depart from a processor and travel to another. From the user's point of view, communication between processors is implemented with special features of the language accepted by the parallel machine, such as special function calls, e.g. *send* and *receive*, or special symbols indicating which processor is the source of the required data. These language features might add some overhead to the implementation, for example copying of data into temporary buffers and vice versa, addressing data and/or buffers, addressing the function calls, etc.

Let us assume a distributed memory machine with some interconnection network. A mathematical model for counting the time spent in communication is given by the formula

$$t(0) = \kappa_1 \ , \quad t(b) = \kappa_i + \pi_i \cdot b \quad \text{if} \quad \beta_{i-1} < b \le \beta_i \ , \qquad \text{for} \quad i = 1,...,s, \tag{1.3}$$

where $t(b)$ is the time spent for b bytes to move between two direct neighbour processors, s is the number of protocols used (often $s = 1$), κ_i, $i = 1,...,s$ are communication overhead constants required for any message (even of 0 bytes) for each protocol, π_i, $i = 1,...,s$ are penalty factors for each protocol, $b_0 = 0$, and β_i, $i = 1,...,s$ are the maximum number of bytes handled by each protocol.

In Table 1.1, for the iPSC/2 [Inte88], $s = 2$, $\kappa_1 = 0.55$, $\kappa_2 = 0.97$, $\pi_1 \simeq .2\times10^{-3}$, $\pi_2 \simeq .36\times10^{-3}$, $\beta_1 = 100$ and $\beta_2 = 16384$. For comparison, a floating-point operation (a multiplication and an addition of floating-point numbers f_i, $i = 1,2,3$ i.e. $f_1 + f_2 \cdot f_3$) takes $f = .137\cdot10^{-1}$ milliseconds on the iPSC/2 hypercube processor. On the iPSC/860 [Bokh90], the time complexity can be approximated by formula (1.3) with $s = 2$, $\kappa_1 = 0.067$, $\kappa_2 = 0.177$, $\pi_1 = .51\cdot10^{-3}$.

Table 1.1. Communication time in milliseconds for several size messages between direct neighbour processors on the iPSC hypercubes.

bytes	0	8	64	100	104	1024	8192	16384
iPSC/2	0.55	0.55	0.56	0.57	1.01	1.34	3.92	6.87
iPSC/860	0.067	0.071	0.100	0.118	0.218	0.576	3.138	6.099

For convenience, in the rest of the paper, we define the penalty factors to be per number, i.e. for the iPSC/2 $\pi_1 \simeq .8\times10^{-3}$, $\pi_2 \simeq .144\times10^{-2}$, since all the messages sent and received are 4 byte integer or floating-point numbers. Thus, $t(\frac{b}{4})$ will be used in place of $t(b)$, and β_i, $i = 1,...,s$ are updated appropriately.

2. Natural ordering and direct solvers.

A very common ordering of the points/elements of Ω is the *natural* ordering, which for two-dimensional problems is bottom-up then left to right (or vice versa). In general, matrices that arise from the application of discretisation methods to d-dimensional BVPs and the natural ordering are banded, with bandwidth proportional to the number of discretisation points in d-1 dimensions, and possess a recursive block structure, where every block has the structure of the matrix corresponding to the problem in d-1 dimensions.

If we apply standard band Gauss elimination (GE) to such matrices, and we assume that a processor is assigned the computation associated with at least one row or column of the matrix, then the degree of parallelism is limited by the number of discretisation points in d-1 dimensions, the

communication requirements include the broadcasting of vectors (rows or columns) from a processor to all other processors at each stage of GE, and load balancing can be bad, unless a sophisticated implementation of send-ahead takes place [Orte88]. Similar problems arise with back-and-forward substitution (b/f/s) algorithms applied to such matrices.

In the next sections, we will study a number of methods for the parallel computation of PDEs, which try to overcome the above problems with quite a success.

3. Natural ordering and iterative solvers.

Some iterative linear solvers require the knowledge of vectors computed at the previous iteration(s) only (*inner-asynchronous*), e.g. Jacobi, Conjugate Gradient (CG), while others require partial knowledge of vectors computed at the current iteration (*inner-synchronous*), e.g. SOR, SSOR, Preconditioned Conjugate Gradient (PCG) with Incomplete Factorisation (IF-PCG) or SSOR preconditioning. In practice, the parallel implementation of inner-synchronous methods for banded systems has the same disadvantages as the b/f/s algorithms for such systems. For the moment, we concentrate our discussion to the inner-asynchronous methods.

At each iteration, these methods require the computation of the product of the discretisation matrix with a vector computed during the previous iteration, the computation of one or more inner products and norms of some vectors and the update of the solution, residual and possibly direction vectors.

The communication requirements during each iteration of inner-asynchronous methods arise during a) the computation of the matrix-vector product and b) the inner product and norm computations. (The updates of vectors do not require communication.) There is an important distinction between these two cases of communication. In the case of matrix-vector product, the communication requirements are dictated by the discretisation stencil and the way the data are assigned to processors. In the case of inner product or norm calculation, the communication requirements are global and do not depend neither on the discretisation method, nor on the assignment of data. In the next subsections, we will study each case separately.

3.1. Matrix-vector multiplication.

Assume a two-dimensional BVP, and a two-dimensional stencil used for the discretisation of the BVP. The stencil shows the dependency of the central point/unknown on other points/unknowns of the stencil. In order to preserve this local dependency, the distribution of data to processors has to follow the geometry of the problem and stencil.

We divide either the geometrical data of the problem, i.e. grid lines, points, elements, etc. or the algorithmical data, i.e. equations, unknowns, into smaller groups constituting local entities. The division reflects the geometry of the BVP (and of the stencil), either in all its dimensions (i.e. it can be a rectangular division), or in fewer dimensions (i.e. it can be a strip division). Assume the matrix is stored by rows. Then each processor will be assigned rows of the matrix corresponding to points or elements belonging to the same subdivision. A processor will also be assigned the respective components of vectors (e.g. solution, right side, residual vectors, etc.)

As an example, let us take the case of a 9-point-star stencil of Figure 1.1(b). Let n and m be the size of the grid discretisation (usually the number of grid points $\pm$ some small constant) in the two (x and y) dimensions respectively. With the strip assignment of data to processors and due to the block tridiagonal structure of the matrix (reflecting the shape of the stencil) any processor needs to receive at most $2m$ (or $2n$) components from neighbour processors. On a hypercube with the processors laied according to standard Gray codes this means neighbour communication. A linear array of processors would perform equally well. The amount of data moved between neighbours is proportional to the

length of the strip. The communication complexity does not depend on the total number of processors.

Assume a processor is assigned a strip of k grid lines, $k \geq 1$. The time spent in communication is $t_{mv}^{b,9p} = 2t(m)$, if we assume blocking send and receive (s/r) operations. For a larger stencil (21-point-star) like Figure 1.1(c), assuming each processor is assigned at least two grid lines of data, the time spent in communication is $t_{mv}^{b,21p} = 2t(2m)$, while if we assume that each processor is assigned one grid line of data, there is second nearest neighbour communication, and the time spent in communication is $t_{mv}^{b,21p} = 4t(m)$. Under certain hardware and software capabilities the communication time can be reduced. For example, with a 9-point-star stencil, the time spent in communication is $t_{mv}^{nb,9p} = 2\max\{t(m) - 3mf, 0\}$ when $k = 1$, $t_{mv}^{nb,9p} = \max\{2t(m) - (9(k-2)mf + 12mf), 0\}$ when $k > 1$ if we assume non-blocking s/r operations and overlapping computation and communication, $t_{mv}^{bd,9p} = t(m)$ if we assume blocking s/r operations and a capability of double links for simultaneous s/r, and $t_{mv}^{nbd,9p} = \max\{t(m) - 3mf, 0\}$ when $k = 1$, $t_{mv}^{nbd,9p} = \max\{t(m) - (9(k-2)mf + 12mf), 0\}$ when $k > 1$, if we assume both non-blocking s/r operations, and a capability of double links for sending and receiving at the same time. If the constant overhead to start the sending of a message, i.e. the κ's, is small compared to the penalty factor per byte, i.e. the π's, and the floating-point operation cost f, then it might worth using element-by-element send-ahead. Then the time is $t_{mv}^{nb,9p} = \max\{2mt(1) - 9kmf, 0\}$, if we assume non-blocking s/r operations.

With a two-dimensional division of the data into $p = p_n \times p_m$ subrectangles, and a similar layout of the processors of a hypercube or of a two-dimensional grid, some second nearest neighbour communication might be required. For example, with a 9-point-star stencil of Figure 1.1(b), the communication time would be $t_{mv}^{b,9p} = t(\frac{n}{p_n}) + t(\frac{m}{p_m}+1) + t(\frac{n}{p_n}+1) + t(\frac{m}{p_m}+2)$, if we assume that the exchange of data is done in the order first north-south, then east-west, then north-south, finally east-west [Chri90b]. In the case of a 5-point-star stencil of Figure 1.1(a) the communication complexity becomes $t_{mv}^{b,5p} = 2t(\frac{n}{p_n}) + 2t(\frac{m}{p_m})$, due to the fact that there are no corner points belonging to second nearest neighbours to be transferred. With non-blocking s/r operations and double links the latter becomes $t_{mv}^{nbd,5p} = \max\{\max\{t(\frac{n}{p_n}), t(\frac{m}{p_m})\} - (\frac{5mnf}{p} - 2\frac{n}{p_n} - 2\frac{m}{p_m}), 0\}$. Note that overlapping of communication and computation to the farthest extent, might require sophisticated implementation, and thus add overhead. All the formulas given above for communication times in the case of non-blocking s/r operations are approximate, as they depend on the implementation and the exact order with which communication calls and computation are done. Also, for simplicity, we ignored the special treatment of the boundary points.

Table 3.1 shows the communication time on the iPSC/2 hypercube for a matrix-vector multiplication, when the matrix arises from a 9-point-star stencil and the s/r operations are blocking. From Table 3.1, we can observe that the communication time does not depend on the number of processors, with the exception of the two-processor case, where only unidirectional communication is required. The communication time depends discontinuously on the grid size. There are jumps whenever the size of the messages exceeds 100 bytes. Other than this, the grid size affects the communication time very little, due to the small per number penalty factors π_1 and π_2.

Table 3.1. Communication time in milliseconds on the iPSC/2 hypercube for the matrix-vector multiplication when the matrix arises from 9-point-star stencil, and for various grid sizes.

$p = p\times1$ grid	2	4	8	16	32
24×24	0.64	1.28	1.28	1.28	1.28
48×48	1.28	2.56	2.88	2.88	2.88
96×96	1.28	2.56	2.88	2.88	2.88

We have used strip divisions so far. The advantage of subrectangle divisions is that we can use more processors, the total amount of data to be transferred is less, and in some cases we can choose p_n and p_m such that the messages are of length less than the threshold β_1. In a later section we show the practical implications of this property.

3.2. Inner product and norm computation.

The computation of the inner product of two vectors consists of first forming the local inner products corresponding to vector components stored locally in a processor, and then summing-up the local inner products. This requires global communication, in a standard fan-in technique. In most cases, it is desirable to have the result reside in all processors. This can be accomplished by either a fan-out of the final result from the processor, in which it resides, to the other processors, which fan-out follows the fan-in summation, or by using a global sum-up, in place of both the fan-in and fan-out. Global sum-up can be advantageous, when double links exist and non-blocking s/r operations are used. In addition, it inherits less loop and indexing overhead. It was found that, on the average, global sum-up was about 1.5 times faster than fan-in followed by fan-out, even if blocking s/r operations are used. The time spent in communication during the global summation is $t_{gs}^{b} = 2(\log_2 p)\, t(1)$ or $t_{gs}^{bd} = (\log_2 p)\, t(1)$.

Table 3.2. Communication time in milliseconds on the iPSC/2 hypercube for summing-up p numbers distributed in the p processors.

p	2	4	8	16	32
global	1	1.5	2.1	2.7	3.3
2-pass	1.5	2.5	3.0	3.7	4.9

In the case of global summation, there is no benefit of using non-blocking s/r operations, because all intermediate results to be sent, depend on previous results to be received. Table 3.2 shows the communication time for fan-in/fan-out (2-pass) and global summation, measured experimentally. The s/r operations are blocking.

3.3. Percentage of communication over total time.

In Table 3.3, we present the experimentally measured and the theoretically expected percentage of communication time over total time for the application of the CG method to a system of 9-point-star equations. The theoretical percentage is given by the formula

$$\frac{2t(m) + 4(log_2 p)t(1)}{2t(m) + 4(log_2 p)t(1) + 9{\cdot}n{\cdot}m + 5{\cdot}n{\cdot}m},$$

where we have taken in account that each CG iteration requires two inner products and updates of three vectors.

Table 3.3. Percentage of communication time over the total time on the iPSC/2 hypercube for a CG iteration on a 9-point-star system. For comparison, the theoretically expected percentage is shown in the second column for each processor configuration.

$p = p\times 1$ grid	2		32	
64×64	1.4%	1.1%	31%	35%
96×96	0.6%	0.5%	17%	19%

4. Multicolour ordering and iterative solvers.

In this section, we study the parallel implementation and communication complexity of inner-synchronous iterative solvers applied to systems arising from the discretisation of BVPs.

At each iteration, these methods require a forward and/or back substitution for the lower and/or upper triangular part of the matrix, a multiplication of the upper and/or lower triangular part of the matrix with a vector computed during the previous iteration, the computation of one or more inner products and norms of some vectors and the update of the solution, residual and possibly direction vectors.

For the natural ordering and two-dimensional BVPs, due to the f/b/s, the dependency of the computations within an iteration step of these solvers allows a degree of parallelism increasing linearly with the size of the problem in one dimension. Following a *multicolour* ordering [Orte88] we can break the dependency of computations within an iteration, in a way so that a high degree of parallelism is achieved. More specifically, the degree of parallelism of synchronous iterative methods increases linearly with the (total) size of the problem divided by the number of colours used. Vector components of the same colour do not depend on each other. Thus the computation of vector components of the same colour can be done asynchronously during each iteration.

The communication requirements of each iteration of inner-synchronous methods arise during the f/b/s, the matrix-vector multiplication, and the inner product and norm computation. The communication for the matrix-vector multiplication is similar to that of Section 3.1. Inner products and norms are computed as in Section 3.2. Back substitution is similar to forward. We concentrate in the communication required for the forward substitution.

4.1. Forward substitution.

The type of communication required for forward substitution (f/s) for the lower part of multicoloured matrices depends on the stencil, on the assignment of data to processors, as well as the multicolouring used. There is usually no unique way to colour the points of a stencil in order to break the dependency between unknowns, even if the minimum number of colours needed can be specified. For a 9-point-star stencil four colours are required and suffice in order to break the stencil dependency. The four colours can be arranged in more than one ways achieving the same goal of decoupling the unknowns. Figure 4.1 shows three four-colour indexings for stencils of type of Figure 1.1(b). These are the only ones, up to permutation of colours.

Indexings (a) and (b) of Figure 4.1 are the one the transpose of the other. From the point of view of communication it should make no difference to use either (a) or (b). Our discussion will be limited to type (a) and (c) four-colour indexings.

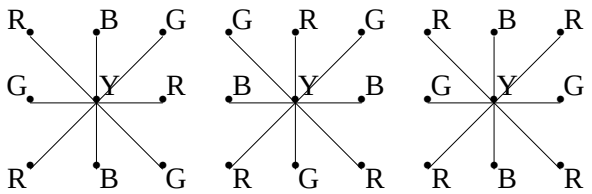

Figure 4.1. Indexing of 9-point-star stencils with four colours, red (R), black (B), green (G) and yellow (Y).

The assignment of data to processors is done by splitting the same coloured points/equations/unknowns as evenly as possibly among the processors. In order to preserve the locality of data transfer, we divide the domain of the BVP (and at the same time the points/equations/unknowns) either into p strips each having an even number of grid lines, or into p subrectangles each having and even number of points in each direction. The computation associated with one strip or subrectangle is assigned to one processor. The corresponding rows of the matrix, the respective right side components and the respective components of any vector required during the computation are assigned similarly.

When the processors are assigned strips, they are arranged in an one-dimensional grid, and the sending of data consists of a unidirectional direct neighbour send. One s/r of size n (or m) is required per processor (of the red and green components if vertical strips are assumed). This is independent of the four-colour indexings (a) and (c) of Figure 4.1. In other words, the communication in the strip assignment case is equivalent to that for a red-black strip colouring (*zebra* colouring [Orte88]). In some methods, e.g. SOR, the f/s is accompanied by a matrix-vector multiplication for the upper triangular part of the matrix. In this case two s/r's are required per processor (one of the red and green components for the f/s and one of the black and yellow components for the matrix-vector multiplication for the upper triangular part of the matrix).

When the processors are assigned subrectangles, and are arranged in a two-dimensional grid, the exchange of data for the f/s is in three directions (and two diagonal ones), independently of the four-colour indexings of Figure 4.1. There is one s/r of size $\frac{m}{p_m}+1$ (or $\frac{n}{p_n}+1$) along each of the two directions, and one s/r of size $\frac{m}{2p_m}$ along the third direction. If the f/s is accompanied by a matrix-vector multiplication for the upper triangular part of the matrix, then we can save in communication by choosing the appropriate indexing. When the four-colour indexing of Figure 4.1(a) is adopted, and if a processor does not have the capability of sending a message to more than one different processors simultaneously, at least two unidirectional direct neighbour s/r's are required for every colour. This totals to at least 8 s/r's. When the four-colour indexing of Figure 4.1(c) is adopted though, and the computation follows the order red-green-yellow-black, i.e. a cyclic way, two s/r's of messages of size $\frac{m}{p_m}+2$ and two s/r's of messages of size $\frac{n}{p_n}+2$ are required per processor, even if a processor does not have the broadcast capability. Thus the time spent in communication during a f/s and a matrix-vector multiplication for the upper triangular part of the matrix is $t_{fs}^{b,\,9p} = 2t(m)$ for one-dimensional layout of processors, and $t_{fs}^{b,\,9p} = 2t(\frac{m}{p_m}+2) + 2t(\frac{n}{p_n}+2)$ for a $p_n \times p_m$ two-dimensional layout of processors. Note that there is no advantage of using double links, since all communication is unidirectional. Some overlapping of computation and communication is possible, and so the use of non-blocking s/r operations may

be advantageous. The above communication complexity for the case of two-dimensional layout of processors does not contradict the results of [Elma89], where the communication of the matrix-vector multiplication for the upper triangular part of the matrix is not considered, as it is not a part of the PCG algorithm.

For some grid sizes, the communication complexity in the case of two-dimensional layout of processors allows us to choose p_n and p_m in a way so that all messages are of length less than β_1. The nice effects of this property are shown in the results of Table 4.1. The one-dimensional layout of processors has the advantage of only 2 s/r's per iteration, instead of 4 of the two-dimensional layout. The two-dimensional processor layout is beneficial in the case that n, m are very large (>> 1000), or when all the 4 messages are below 100 bytes. This agrees well with the conditions arising by studying the inequality $2t(m) \geq 2t(\frac{m}{p_m}+2) + 2t(\frac{n}{p_n}+2)$. The theoretically expected communication complexity agrees well with the experimentally measured one. Note that the experimentally measured times shown in the second column of data for each processor layout of Table 4.1 include the execution of the s/r operations, the required indexing, as well as some copying of data into temporary storage. In fact, this cannot be avoided in the case of two-dimensional division, because the points/components in one of the two directions will not be stored consecutively, if we follow a natural ordering inside each subrectangle and for each colour. There is also copying overhead in order to combine the red (green) and black (yellow) colours into one s/r. The overhead due to copying into temporary storage can be minimised if the ordering of each colour inside each subrectangle and the layout of processors are chosen appropriately. Thus, when the two-dimensional layout of processors is adjusted so that the size of all 4 messages is equal to or below 25 numbers, i.e. 100 bytes, and the ordering is chosen so that the copying is done along the shortest boundary of each processor, we get the lowest communication overhead. An example of the effects of careful choice of ordering and processor layout is shown in the case of 32 processors and 96×96 grid, in which the lowest communication overhead is obtained for 8×4 processor layout. Note that the copying and restoring of a floating-point number on the iPSC/2 takes about as much time as a floating-point operation.

Table 4.1. Communication time in milliseconds on the iPSC/2 hypercube during a forward substitution and an upper-triangular matrix-vector multiplication for a four-colour indexing of Figure 4.1(c) for various grid sizes and processor layouts. For comparison, the theoretically expected communication complexity is shown in the second column for each processor layout. The copying overhead is taken in account.

processors	2		4		8		16				32					
layout	2×1		4×1		8×1		8×2		16×1		8×4		16×2		32×1	
grid																
48×48	1.28	1.37	3.20	2.74	3.20	2.74	2.56	2.66								
64×64	1.60	1.50	3.20	3.00	3.20	3.00	4.16	3.70	3.20	3.00	2.88	2.68	4.16	3.64	3.52	3.00
96×96	1.60	1.77	3.20	3.53	3.20	3.53	4.16	4.02	3.84	3.53	2.88	2.92	4.16	3.93	3.52	3.00

5. Domain decomposition ordering.

Domain Decomposition (DD) methods assume the partition of the domain of problem definition into non-overlapping subdomains. With the partitioning of the domain into subdomains we indirectly achieve the division of the geometric constructs of the domain, i.e. points or elements of the grid, as well as the algorithmic constructs, i.e. rows and columns of the discretisation matrix, into subsets. Some of points/elements lie in the (interior of the) subdomains and some lie on or constitute the interfaces between subdomains. The necessary minimal width of the interfaces, required to break the

dependency of interior of subdomains on each other, depends upon the discretisation stencil used. For 9-point-star, or 5-point-star stencils for example, a single line of data suffices. For larger stencils, more than one line of data is required, for example for the stencil of Figure 1.1(c), two lines of data are required and suffice.

If we order the interface points/elements first, followed by the points/elements interior to subdomains, for each subdomain in order, and apply the same ordering to the algorithmic constructs i.e. rows (equations) and columns (unknowns) of the discretisation matrix, the result is the so called *arrowhead matrix* [Orte88]. That is, the discretised PDE takes the form of the 2×2 block system

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}, \tag{5.1}$$

where A_{00} represents the relation between interface equations/unknowns, A_{11} the relation between subdomains' equations/unknowns and A_{01}, A_{10} represent the interaction between subdomains and interfaces. A_{11} is block diagonal due to the independency of interiors of subdomains between each other.

Systems like (5.1) are often solved by the Schur complement method, which is equivalent to a block Gauss elimination applied to (5.1). The Schur complement method involves the elimination of the interior to subdomains unknowns and the solution of the so called *capacitance* or *Schur Complement* (SC) system $Cx_0 = w$, where $C = A_{00} - A_{01}A_{11}^{-1}A_{10}$ and $w = b_0 - A_{01}A_{11}^{-1}b_1$. The solution of the capacitance system is often carried out by iterative methods, like PCG, in order to avoid the explicit calculation of the entries of C and thus of the entries of $A_{11}^{-1}A_{10}$.

From the partitioning of the data in the DD method, it is made clear that the computation corresponding to interior of subdomains can be carried out independently in different processors. The computation corresponding to interfaces requires interaction between processors though. In order to preserve the locality of data transfer we assign the north and east to each subdomain interface to the respective subdomain/processor. Thus, a processor is assigned the data and computation corresponding to one (or more) subdomains including its (north and east) interface. This means that each processor will be assigned one (or more) blocks of A_{11}, the respective rows of A_{10}, the rows of A_{00} and A_{01} corresponding to its interface equations, and all respective components of vectors required through the computation.

The communication requirements of DD methods arise when the processing of interface equations happens. More specifically, if an iterative solver is applied to the capacitance system, the product of C with a given vector has to be computed, a global stopping criterion to be checked (this means calculation of a norm), and possibly additional inner products to be computed (for CG methods). The type of communication required for the matrix-vector multiplication is similar to that discussed in Section 3.1, and of that required for norms and inner products is discussed in Section 3.2. We should mention though, that the above assignment of data to processors is not fully balanced. A fully balanced assignment would split the interface points as equivalently as possible among processors, but would require global communication, and thus would give us a communication complexity depending on the number of processors, as well as the amount of data transfer. In most cases, the slight improvement of balance in data and computational load with this assignment, does not worth the sacrifice of locality of data transfer [Chri89], [Chri91].

6. Schwarz splitting methods.

Schwarz Splitting (SS) methods assume the partition of the domain of problem definition into overlapping subdomains, so that each subdomain has a fictitious boundary, which belongs to the interior of the neighbour subdomains. A generic algorithm for the SS methods loops repeatedly over all

subdomains, each time updating the fictitious boundary conditions of a subdomain, using neighbour subdomains's solutions, until some stopping criterion is satisfied.

From the partitioning of the data in the SS method, it is made clear that the computation corresponding to each subdomain can be carried out independently in different processors (assume the additive version of SS method). The communication requirements of SS methods arise between iterations of the SS method, when the approximation on the fictitious boundary has to be passed to the neighbour processors, and when a norm is calculated to implement the stopping criterion test. The calculation of a norm is already discussed in Section 3.2. The fictitious boundary results in neighbour communication similar to the communication required for inner-asynchronous iterative solvers and for DD methods discussed in Sections 3.1 and 3.5. There are two advantages though for the SS method. 1) The number of data to be transferred does not depend on the discretisation stencil. Only one line of data needs to be transferred to each neighbour, the data on the fictitious boundary. (This holds for the simple SS method described in this section, but it is not true when a SS preconditioner is used.) 2) The computation of the SS method does not depend on an inner product or norm. The norm calculation needs to be carried out occasionally and not necessarily at each iteration. (This is also true for iterative solvers like SOR but not for CG methods.)

7. Parallel discretisation of the BVP problem.

The computation of the entries of the matrix and of the right side vector assigned to a processor is carried out locally on the processor. If a discretisation method like Galerkin is used, some local nearest neighbour communication might be required to sum up the neighbour integrals. Other discretisation methods like finite difference and standard collocation are in general totally asynchronous. Two-step collocation methods [Hous88a], [Hous88b], [Irod88], [Chri90a] require local communication in order to compute the update of the right side vector for the second step. Thus two-step collocation methods are equivalent to Galerkin with respect to communication. Same holds for some high order finite difference methods. Summarising, discretisation methods may require local communication similar to that discussed in Section 3.1.

8. Conclusions.

We have considered the parallel implementation of a number of methods for the parallel computation of elliptic PDEs in d space dimensions. The communication requirements of these methods fall into two categories: local (in d or fewer dimensions) and global. The local communication requirements do not depend on the total number of processors, but they depend on the size of discretisation grid along the direction of transfer, and, except few exceptions, on the discretisation stencil. The global communication requirements depend only on the total number of processors. The table below summarises the results.

Table 8.1. Communication requirements for various PDE discretisation methods and linear solvers.

type solver	local ††	global		discret. method	local
Jacobi/natural	matrix-vector mult.	norm*		Galerkin	summation
SOR/multicol.	forward substitution	norm*		Finite Difference	-
CG/natural	matrix-vector mult.	norm*	inner prod.	HODIE	summation
IFCG/multicol.	forward substitution	norm*	inner prod.	Collocation	-
DD-PCG	matrix-vector mult.	norm*	inner prod.	Two-step colloc.	summation
SS	fictitious boundary†	norm*			

† amount of data transfer not affected by the discretisation stencil

†† locality reflects the geometry of the problem in all its dimensions or in fewer dimensions

* may be performed occasionally

References.

[Ames77] **Ames, W. F.,** *Numerical methods for partial differential equations,* Academic Press 1977.

[Bokh90] **Bokhari, Shahid,** *Communication overhead on the Intel iPSC/860 hypercube.* ICASE Interim Report 10, May 1990, NASA Langley Research Center, Hampton, VA, U.S.A. (14 pgs).

[Chri89] **Christara, C. C.** and **E. N. Houstis,** *A domain decomposition spline collocation method for elliptic PDEs,* Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications (HCCA4), March 1989, Monterey, CA, U.S.A., pp. 1267−1273.

[Chri90a] **Christara, C. C.,** *Quadratic spline collocation methods for elliptic PDEs,* submitted for publication (32 pgs).

[Chri90b] **Christara, C. C.,** *Schur complement PCG methods for spline collocation equations,* Proceedings of the 1990 International Conference on Supercomputing (ICS90), June 1990, Amsterdam, the Netherlands, pp. 108−120.

[Chri91] **Christara, C. C.,** *Domain Decomposition and Incomplete Factorisation Methods for PDEs,* Proceedings of the 6th Distributed Memory Computing Conference (DMCC6), April−May 1991, Portland, OR, U.S.A., pp. 369−377.

[Elma89] **Elman, H. C.** and **E. Agron,** *Ordering techniques for the preconditioned conjugate gradient method on parallel computers,* Comput. Phys. Comm., **53** (1989), pp. 237–252.

[Grop88] **Gropp, W. D.** and **D. E. Keyes,** *Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations* SIAM J. Sci. Statist. Comput., **9, 2** (1988), pp. 312–326.

[Gust88] **Gustafson, J. L., G. R. Montry** and **R. E. Benner,** *Development of Parallel Methods for a 1024-Processor Hypercube,* SIAM J. Sci. Statist. Comput., **9,** 4 (1988), pp. 609–638.

[Hous88a] **Houstis, E. N., C. C. Christara** and **J. R. Rice,** *Quadratic spline collocation methods for two-point boundary value problems,* Internat. J. Numer. Methods Engrg., **26,** (1988), pp.

935−952.

[Hous88b] **Houstis, E. N., E. A. Vavalis** and **J. R. Rice,** *Convergence of an $O(h^4)$ cubic spline collocation method for elliptic partial differential equations,* SIAM J. Numer. Anal., **25** (1988), pp. 54−74.

[Inte88] **Intel Scientific Computers,** *iPSC/2 Performance report,* January 1988, Beaverton, OR, U.S.A.

[Irod88] **Irodotou-Ellina, M.** and **E. N. Houstis,** *An $O(h^6)$ quintic spline collocation method for fourth-order two-point boundary value problems,* BIT, **28** (1988), pp. 288−301.

[Keye87] **Keyes, D. E.,** and **W. D. Gropp,** *A comparison of domain decomposition techniques for elliptic PDEs and their parallel implementation,* SIAM J. Sci. Statist. Comput., **8** (1987), pp. s166−s202.

[Lync80] **Lynch, R. E.** and **J. R. Rice,** *A High-Order Difference Method for Differential Equations,* Math. Comp., **34** (1980), pp. 333−372.

[Orte88] **Ortega, J. M.,** *Introduction to Parallel and Vector Solution of Linear Systems,* Plenum Press, 1988 (305 pgs).