Numerical Solution of the Shallow-Water Equations on Distributed Memory Systems

by

Xiaoliang Ding

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

© Copyright by Xiaoliang Ding 1998

Contents

List of Tables

List of Figures

Abstract

The shallow-water equations are often used as a mathematical model when numerical methods for solving weather or climate prediction problems are tested. This thesis studies the performance and scalability of numerical methods for the shallow-water equations on distributed memory systems. Time integration of the numerical methods is based on a two-time-level semi-implicit semi-Lagrangian scheme. To solve the Helmholtz problem that arises at each time-step, a fast direct solver based on FFTs is used. This technique requires a data transposition, which is a time consuming operation on a distributed memory system.

The data transposition requires an all-to-all type communication. The Direct Exchange algorithm, commonly used for this task, is inefficient if the number of processors is large. We investigate a series of more sophisticated techniques: the Ring Exchange, Mesh Exchange and Cube Exchange algorithms. We compare the performance of these techniques with that of the Message Passing Interface collective communication routine. We perform an isoefficiency analysis to measure the scalability of the parallel system. The numerical experiments are carried out on a Cray T3E with 512 processors and on an ethernet-connected cluster of 36 workstations.

Both the analysis and our numerical results indicate that the Cube Exchange and Mesh Exchange algorithms perform better than the other methods. However, even with the more sophisticated techniques, the parallel algorithm for the shallow-water equations remains essentially unscalable.

Acknowledgments

I wish to thank my thesis advisors, Professors Ken Jackson and Christina Christara, for their support during my years at the University of Toronto. They were always available for advice, comments, constructive criticism and encouragement.

I would also like to thank Dr. Steve Thomas, who provided the shallow-water code. He also contributed much to the thesis with his comments and advice, particularly on the overall direction of the research.

I greatly appreciate access to the Cray T3E machine provided by Cray Research Inc.

Finally I want to express my gratitude to my daughter Jiayin and my wife Yaohua for their love, support and encouragement.

Chapter 1

Introduction

An *atmospheric computer model* is a computer program that simulates atmospheric processes (wind, clouds, precipitation, etc.) that influence weather or climate. It may be used to predict the weather, or to study the impact on climate of effects such as the increased concentration of atmospheric carbon dioxide. The computer model uses numerical methods to solve a set of partial differential equations, describing the basic fluid dynamical behavior of the atmosphere. The computer model performs a time integration scheme to determine the state of the atmosphere at the future time, based on the initial state. It is well known that weather forecasts require a large amount of computing power. Computers with sustained teraflop performance and gigabyte capacity memories are needed for such a computation. Such capability can be achieved only through the use of hundreds to thousands of processors; thus the development of a new model for such architectures is a principle goal of several research groups. (See for example [17]).

The shallow-water equations are often used to test new numerical methods, since they are a two-dimensional prototype of the three-dimensional models that govern atmospheric states [18, 19]. The shallow-water equations exhibit the major difficulties associated with the horizontal dynamical aspects of atmospheric modeling on the spherical earth. They also provide a first test to weed out potentially non-competitive schemes without the effort of building a complete model. Therefore, they are often used to evaluate numerical methods proposed for atmospheric modeling and to identify the potential trade-offs which must always be made in numerical modeling.

A long-standing problem in the integration of atmosphere models is that the maximum time step allowed in many numerical methods is governed by stability considerations rather than accuracy. Early atmosphere models used explicit schemes, whose time step is limited by the propagation speed of gravity waves. By associating a semi-Lagrangian treatment of advection with a semi-implicit treatment of gravity waves, longer time steps can be used without degrading the accuracy of the solution [10, 11]. In this thesis, we investigate the parallelization of the semi-implicit semi-Lagrangian shallow-water model. The semi-implicit scheme is formulated as an implicit correction of the explicit scheme. As a result, we have to solve a two-dimensional Helmholtz equation at every time step. An elegant and effective means of solving the Helmholtz equation is to employ FFT techniques in one direction followed by tridiagonal solves in the other direction. This is called a Fast Direct Solver [18]. Depending on the direction of the data dependence within the Direct Solver, the grid columns or rows are distributed to the processors. In this way, we can perform each FFT and each tridiagonal system solution locally. However, the transposition of data is required, each time the direction of data dependence changes. For parallel computation, this implies that each processor has to communicate with each other to redistribute the data. This is a potentially time-consuming task for parallel architectures [7, 9]. To improve the parallel performance, several alternative transposition algorithms are proposed in this thesis. The parallel model is implemented using MPI (Message Passing Interface), and the numerical experiments are carried out on a Cray T3E machine and a set of workstations connected via an ethernet.

Chapter 2 introduces the concept of the parallel system and the terminologies used in the parallel performance analysis. Chapter 3 describes the shallow-water model and the sequential numerical algorithms. Chapter 4 describes the parallelization of the shallow-water model and the global transpose algorithms. In Chapter 5, numerical results and comparisons of the algorithms are presented. Conclusions are presented in Chapter 6.

Chapter 2

Parallel Systems

A sequential application is usually evaluated in terms of its execution time, expressed as a function of the problem size. The execution time of a parallel application, however, depends not only on the problem size, but also the architecture of the parallel computer and the number of processors. Hence it is necessary to consider both the application and the parallel computer when measuring the system performance. A **parallel system** is defined as the combination of an application and the computer system on which the application runs.

2.1 A General Parallel Machine Model

Parallel machines can be characterized in many ways that distinguish one from another. In term of the control mechanism, processing units in parallel computers either operate under the centralized control of a single control unit or work independently. A **single instruction stream, multiple data stream (SIMD)** has one single control unit which dispatches the same instructions to each processor. In a **multiple instruction stream, multiple data stream (MIMD)** computer, each processor is capable of executing a different program independent of the other processors.

Parallel computers also differ in memory organization. In **distributed-memory** parallel computers, each processor has its own memory, called local or private memory. The **shared-memory** architecture provides hardware support for read and write access by all processors to a shared address space. Processors interact by modifying data objects stored in the shared address space.

Several parallel programming paradigms are commonly used in parallel applications. **Data Parallelism** refers to the concurrency that is obtained when the same operation is applied to some or all elements of a data ensemble. A data-parallel program is a sequence of such operations. In a **Shared Memory** programming model, tasks share a common address space, in which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared memory. **Message Passing** is one of the most widely used parallel programming models today. In this model, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages.

2.2 Performance Metrics of a Parallel System

We define the **execution time** of a parallel program as the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution. On each processor, the execution time is spent on computing, communicating and idling.

The time taken to communicate a message between two processors in the network is called **communication latency**. Communication latency is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principle parameters that determine the communication latency are as follows.

1) **Startup time** (t_s). The startup time is the time required to handle the message by the sending processor. This includes the time to prepare the message and the time to execute the routing algorithm. This delay is incurred only once for a single message transfer.

2) **Per-hop time** (t_h). After a message leaves a processor, it takes a finite amount of time to reach the next processor in its path. The time taken by the head of a

message to travel between two directly-connected processors in the network is called the per-hop time.

3) **Per-word transfer time** (t_w). If the channel bandwidth is r words per second then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time.

Many factors influence the communication latency of a network, such as the topology of the network and the switching techniques.

A *worm-hole routing* technique is widely used in parallel machines. A message being communicated travels in small units called flow-control digits or *flits*. In worm-hole routing, flits are pipelined through the communication network. As soon as a flit is received at an intermediate processor, the flit is passed to the next processor.

Consider a message that traverses l links, and t_h is the per-hop time, then the header of the message takes lt_h time to reach the destination. If the message size is m words, then the entire message will arrive in time mt_w after the arrival of the header of the message. Therefore, the total communication time for worm-hole routing is given by

$$t_s + lt_h + mt_w. \tag{2.1}$$

In a modern parallel machine, the interconnect topology is usually a high dimensional mesh (a 3D Torus in the Cray T3E), or a multistage interconnect (as in the IBM SP2). As a result, the number of links to be traversed from one node to another node is small. Also usually $t_h \ll t_w$, therefore the cost of traversing the links is negligible if the number of links remains small. Therefore the cost of sending a message of size m between two processors can be approximated by

$$t_s + mt_w. \tag{2.2}$$

The notion of **Speedup** is perhaps the earliest metric that has been widely used as a measure of scalability. The speedup S_p of a parallel system is defined as the ratio of the execution time T_1 of a given application run on a single processor to the time T_p taken to solve the same problem on p processors:

$$S_p = \frac{T_1}{T_p}. \tag{2.3}$$

Efficiency is a measure of the fraction of time for which a processor is usefully employed. The efficiency E_p is defined as the ratio of speedup S_p to the number of processors:

$$E_p = \frac{S_p}{p}. \tag{2.4}$$

Ideally, $S_p = p$, and $E_p = 1$. In practice, a parallel system containing p processors does not usually achieve a speedup of p, because part of the time required by the processors is spent on communication or idling. That is, usually $0 < S_p < p$ and $0 < E < 1$, depending on the degree of effectiveness with which the processors are utilized. Figure 2.1 shows a typical relationship between speedup and the number of processors used.

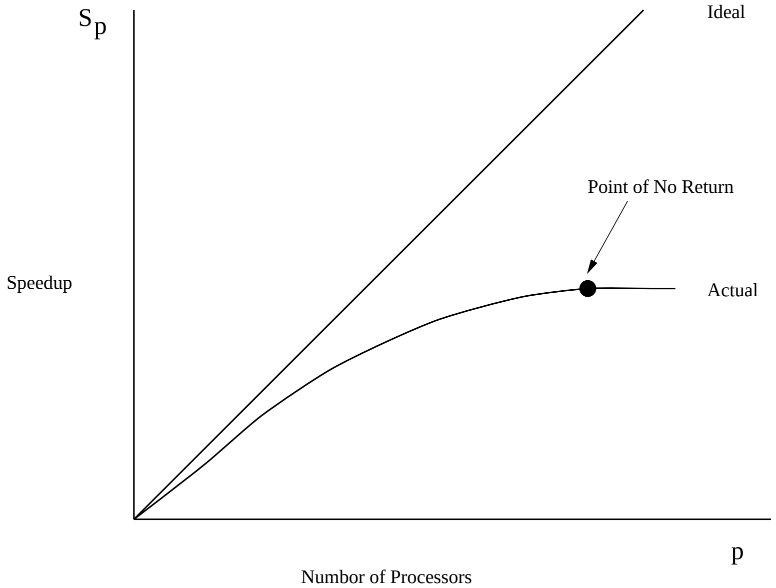

Figure 2.1: Plot of speedup S_p versus the number of processors p for a fixed problem size.

Figure 2.1 shows that the speedup does not increase linearly with the number of processors; instead, it tends to saturate. In other words, the efficiency drops as the number of processors increases. This is often referred to as **Amdahl's law**.

On the other hand, the speedup (efficiency) usually increases as the problem size increases on the same number of processors. If increasing the number of processors reduces efficiency, and increasing the problem size increases efficiency, we might be able to keep efficiency constant by increasing both the number of processors and the problem size simultaneously. The question is: at what rate should we increase the problem size with respect to the number of processors to keep the efficiency fixed? An **isoefficiency function** is used to quantitatively determine the degree of **scalability** of a parallel system, that is, the system's ability to utilize increasing resources efficiently. Before defining the isoefficiency function, we need to introduce the problem size and the overhead function.

We define the **problem size** W as the number of basic operations required to solve the problem on a single processor. Usually we define W as the number of basic operations needed by the best sequential algorithm. If the time taken by the basic operation is t_c on a single processor, then the time complexity of the sequential algorithm is $T_1 = t_c W$, To simplify the notation, we use the normalization $t_c = 1$, so $T_1 = W$.

The parallel execution time can be expressed as a function of the problem size, the overhead function and the number of processors. We define **total overhead** or the **overhead function** of a parallel system as the part of its cost that is not incurred by the sequential algorithm on a single processor. It is the total time collectively spent by all processors in addition to that required by the sequential algorithm to solve the same problem on a single processor. We denote the overhead function of a parallel system by T_o. T_o is a function of the problem size W and the number of processors p, and is often written as $T_o(W, p)$. The total time spent by all processors is pT_p. W units of this time are spent performing useful work, and the remainder is the overhead. That is,

$$T_o(W, p) = pT_p - W \tag{2.5}$$

The overhead incurred in a parallel system includes communication costs, idle time and redundant computations. From the definition of speedup, we have,

$$S_p = \frac{T_1}{T_p} = \frac{W}{\frac{W+T_o(W,p)}{p}} = \frac{Wp}{W + T_o(W,p)}. \tag{2.6}$$

So the efficiency can be written as

$$E_p = \frac{S_p}{p} = \frac{W}{W + T_o(W,p)} = \frac{1}{1 + T_o(W,p)/W}. \tag{2.7}$$

For a desired value of efficiency E, equation (2.7) can be written as

$$W = \frac{E}{1-E} T_o(W,p). \tag{2.8}$$

Let $K = E/(1-E)$ be a constant depending on the efficiency to be maintained. Since $T_o(W,p)$ is a function of W and p, the above equation can be rewritten as

$$W = KT_o(W,p). \tag{2.9}$$

From this equation, the problem size can usually be obtained as a function of p by algebraic manipulations. This function is called **isoefficiency function** of the parallel system. For scalable parallel systems, efficiency can be maintained at a fixed value if the ratio $T_o(W,p)/W$ is maintained at a constant value.

The isoefficiency function determines how easily a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processors. If an isoefficiency function grows slowly with the number of processors, then small increments in the problem size are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. If an isoefficiency function grows rapidly with the number of processor, then the parallel system scales poorly. For an unscalable parallel systems, the isoefficiency function does not exist, because in such systems the efficiency can not be kept at a constant value as p increases, no matter how fast the problem size is increased.

Chapter 3

Solving the 2D Shallow-Water Equations

3.1 The 2D Shallow-Water Equations

The shallow-water equations,

$$\frac{\partial u}{\partial t} + (\mathbf{V} \cdot \nabla)u - fv + \frac{\partial \phi}{\partial x} = 0 \tag{3.1}$$

$$\frac{\partial v}{\partial t} + (\mathbf{V} \cdot \nabla)v + fu + \frac{\partial \phi}{\partial y} = 0 \tag{3.2}$$

$$\frac{\partial \phi}{\partial t} + (\mathbf{V} \cdot \nabla)\phi + \phi \nabla \cdot \mathbf{V} = 0 \tag{3.3}$$

model the inviscid flow of a thin layer of fluid in two space dimensions, where $\mathbf{V} = (u, v)$, u represents the horizontal velocity, v the vertical velocity. Here, $\phi = gh$ is the geopotential height, g is the acceleration due to gravity, h is the height of the free surface of the fluid above the flat bottom, and $f = 1.428 \times 10^{-4} sec^{-1}$ is the Coriolis parameter. The gradient operator ∇ is defined as

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{pmatrix}^T \tag{3.4}$$

in the 2D case.

The inner product is denoted by '$\cdot$', so the advection operator is defined by

$$\mathbf{V} \cdot \nabla = u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y}. \tag{3.5}$$

The problem is defined on a $[0, L_1] \times [0, L_2]$ domain with periodic boundary conditions.

Let $\phi = \bar{\phi} + \tilde{\phi}$, where $\bar{\phi}$ represents the equilibrium geopotential height (constant) and $\tilde{\phi}$ represents the perturbation term. Also let $D = \nabla \cdot \mathbf{V} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$ be the divergence. Using the total derivative $d/dt = \partial/\partial t + (\mathbf{V} \cdot \nabla)$ and noting that $\frac{d}{dt} ln(\frac{\phi}{\bar{\phi}}) = \frac{d}{dt} ln(\phi) = \frac{1}{\phi}\frac{d\phi}{dt}$, we can write (3.1) — (3.3) in an advective form as

$$\frac{du}{dt} - fv + \phi_x = 0 \tag{3.6}$$

$$\frac{dv}{dt} + fu + \phi_y = 0 \tag{3.7}$$

$$\bar{\phi}\frac{d}{dt} ln(\frac{\phi}{\bar{\phi}}) + \bar{\phi}D = 0. \tag{3.8}$$

Given the initial basic state $u = 0$, $v = 0$ and $\bar{\phi} = g\bar{h}$, linearization of equations (3.6)-(3.8) followed by Fourier analysis reveals that solutions exist having phase speeds $c = 0$ and $c = \pm(g\bar{h} + f^2/\phi^2)^{1/2}$, with $\phi^2 = \phi_x^2 + \phi_y^2$. The former represents slow movements of the atmosphere, related to the propagation of Rossby waves, whereas the later are fast-moving gravity waves [10].

3.2 Semi-Implicit Semi-Lagrangian Discretization

For numerical integration by an Eulerian leapfrog scheme, the above analysis suggests that a time step would be limited by the speed of the fast-moving gravity waves. In atmospheric models, it is known that gravity waves propagate at a speed many times larger than that of the Rossby waves, implying that time steps would be many times smaller than those required for an explicit treatment of passive advection. To overcome this limitation, Robert [11] introduced a semi-implicit scheme which amounts to time-averaging the Coriolis and pressure gradient terms in (3.6) and (3.7) and the divergence term in (3.8). So, unless an accurate representation of gravity waves is important, the time step can be increased significantly.

The basic idea of the semi-Lagrangian method can be illustrated by applying it to the one-dimensional advection equation

$$\frac{dF}{dt} = \frac{\partial F}{\partial t} + \frac{dx}{dt}\frac{\partial F}{\partial x} = 0 \tag{3.9}$$

where

$$\frac{dx}{dt} = \mathbf{V}(x,t) \tag{3.10}$$

and $\mathbf{V}(x,t)$ is a given function. Equation (3.9) states that the scalar F is constant along a fluid path (or a trajectory or a characteristic). In Figure 3.1, the *exact* trajectory in the x-t plane of the fluid particle that arrives at mesh point x_m at time t is denoted by the solid curve AC, and an *approximate* straight-line trajectory by the dashed line $A'C$. Assume that we know $F(x,t)$ at all mesh points x at time $t - \frac{\Delta t}{2}$. The idea of semi-Lagrangian advection is to approximately integrate (3.9) along the approximate fluid trajectory $A'C$. Thus,

$$\frac{F(x_m,t) - F(x_m - \alpha, t - \Delta t)}{\Delta t} = 0. \tag{3.11}$$

Here α is the distance $A'D$ the particle travels in x in time Δt, when following the approximate space-time trajectory $A'C$. Thus, if we know α, then the value of F at the arrival point x_m at time t is just the value at the upstream point $x_m - \alpha$ at time $t - \Delta t$. However, we have not yet determined α; and we only know F at mesh points at previous time step $t - \Delta t$. To determine α, note that $\mathbf{V}$ evaluated at the point B of Fig. (3.1) is just the inverse of the slope of the straight line $A'C$, and this gives the following $O(\Delta t^2)$ approximation to equation (3.10)

$$\alpha = \Delta t \mathbf{V}(x - \frac{\alpha}{2}, t - \frac{\Delta t}{2}). \tag{3.12}$$

The velocity $\mathbf{V}$ at the upstream midpoint $(x-\frac{\alpha}{2}, t-\frac{\Delta t}{2})$ is found by linear interpolation between grid points:

$$\mathbf{V}(x, t-\frac{\Delta t}{2}) = \frac{3}{2}\mathbf{V}(x, t-\Delta t) - \frac{1}{2}\mathbf{V}(x, t-2\Delta t). \quad (3.13)$$

To compute the particle displacement α, an iterative scheme is used based on the following equation

$$\alpha^{[k+1]} = \Delta t \mathbf{V}(x - \frac{\alpha^{[k]}}{2}, t - \frac{\Delta t}{2}) \quad (3.14)$$

using the value of α at the previous time step as initial guess for $\alpha^{[0]}$.

The above method can also be applied to the 2D shallow-water equations. By treating the linear terms responsible for these oscillations in an implicit manner, it is possible to lengthen the time step by a factor of 6, at little additional cost and without degrading the accuracy of the solution [14]. Further relaxation of the time step restriction is achieved by combining semi-Lagrangian treatment of advection with a two-time-level semi-implicit treatment of terms responsible for gravitational waves. Using the two-time-level scheme and averaging the Coriolis and pressure gradient terms and the divergence term, the shallow-water equations (3.6)—(3.8) can be discretised as

$$\frac{u^t - u^{t-\Delta t}}{\Delta t} - f\frac{v^t + v^{t-\Delta t}}{2} + \frac{\phi_x^t + \phi_x^{t-\Delta t}}{2} = 0 \quad (3.15)$$

$$\frac{v^t - v^{t-\Delta t}}{\Delta t} + f\frac{u^t + u^{t-\Delta t}}{2} + \frac{\phi_y^t + \phi_y^{t-\Delta t}}{2} = 0 \quad (3.16)$$

$$\frac{[\bar{\phi} ln(\frac{\phi}{\bar{\phi}})]^t - [\bar{\phi} ln(\frac{\phi}{\bar{\phi}})]^{t-\Delta t}}{\Delta t} + \frac{[\bar{\phi}D]^t + [\bar{\phi}D]^{t-\Delta t}}{2} = 0. \quad (3.17)$$

where $[\cdot]^{t-\Delta t}$ represents upstream interpolation in space at time $t-\Delta t$. By moving the upstream terms to the right sides, we obtain the time-discretised approximations to equations (3.6)—(3.8)

$$u - \frac{\Delta t}{2} f v + \frac{\Delta t}{2}\phi_x = R_u \quad (3.18)$$

$$v + \frac{\Delta t}{2} f u + \frac{\Delta t}{2} \phi_y = R_v \tag{3.19}$$

$$\bar{\phi} ln(\frac{\phi}{\bar{\phi}}) + \frac{\Delta t}{2} \bar{\phi} D = R_\phi \tag{3.20}$$

where the right-hand-sides R_u, R_v and R_ϕ are

$$R_u = [u + \frac{\Delta t}{2} f v - \frac{\Delta t}{2} \phi_x]^{t-\Delta t} \tag{3.21}$$

$$R_v = [v - \frac{\Delta t}{2} f u - \frac{\Delta t}{2} \phi_y]^{t-\Delta t} \tag{3.22}$$

$$R_\phi = [\bar{\phi} ln(\frac{\phi}{\bar{\phi}}) - \frac{\Delta t}{2} \bar{\phi} D]^{t-\Delta t}. \tag{3.23}$$

Approximations to R_u, R_v, R_ϕ at the departure point $(\mathbf{x} - \alpha, t - \Delta t)$ are based on an iterative scheme proposed by Robert [11] for computing the particle displacement α, $\alpha^{[k+1]} = \Delta t \mathbf{V}(\mathbf{x} - \alpha^{[k]}/2, t - \Delta t/2)$ where $\mathbf{x}$ is the grid point of a regular Cartesian mesh.

The horizontal momentum equations (3.18) and (3.19) can be written as

$$\begin{bmatrix} 1 & -\frac{\Delta t}{2} f \\ \frac{\Delta t}{2} f & 1 \end{bmatrix} \mathbf{V} = -\frac{\Delta t}{2} \nabla \phi + \begin{bmatrix} R_u \\ R_v \end{bmatrix} \tag{3.24}$$

Following the notation adopted in [18], define

$$\begin{bmatrix} 1 & -\frac{\Delta t}{2} f \\ \frac{\Delta t}{2} f & 1 \end{bmatrix} = \begin{bmatrix} a & b \\ -b & a \end{bmatrix}, \quad \lambda_0 = (\frac{2}{\Delta t})^2 \frac{1}{\bar{\phi}}.$$

Taking the divergence of the momentum equations and then eliminating the divergence term D in the continuity equation, we obtain a nonlinear Helmholtz equation for ϕ

$$\nabla^2 \phi - \lambda_0 \bar{\phi} ln \left(\frac{\phi}{\bar{\phi}} \right) = R_H \tag{3.25}$$

where

$$R_H = \frac{2}{\Delta t}\nabla \cdot (\begin{bmatrix} a & b \\ -b & a \end{bmatrix}\begin{bmatrix} R_u \\ R_v \end{bmatrix}) - \lambda_0 R_\phi. \tag{3.26}$$

After obtaining ϕ by solving the Helmholtz equation, which will be described in the next section, the velocity can be computed from the momentum equations (3.24).

3.3 Solving the Helmholtz Problem

Since $\phi = \bar{\phi} + \tilde{\phi}$ and $\bar{\phi}$ is a constant, Eqn. (3.25) is the same as

$$\nabla^2\tilde{\phi} - \lambda_0\bar{\phi} ln\left(\frac{\phi}{\bar{\phi}}\right) = R_H. \tag{3.27}$$

To use the fixed-point iteration method, we rewrite (3.27) as

$$\nabla^2\tilde{\phi} - \lambda_0\tilde{\phi} = R_H + \lambda_0\bar{\phi}\left[ln\left(\frac{\phi}{\bar{\phi}}\right) - \frac{\tilde{\phi}}{\bar{\phi}}\right]. \tag{3.28}$$

The nonlinear Helmholtz equation (3.25) is solved by using the iteration

$$\nabla^2\tilde{\phi}^{[k+1]} - \lambda_0\tilde{\phi}^{[k+1]} = R_H + \lambda_0\bar{\phi}\left[ln\left(\frac{\phi}{\bar{\phi}}\right) - \frac{\tilde{\phi}}{\bar{\phi}}\right]^{[k]}. \tag{3.29}$$

For simplicity of notation, $\tilde{\phi}$ is replaced by ϕ from here on. Spacial discretization of equation (3.29) is based on an expansion of the geopotential ϕ in terms of piecewise constant element basis function on a regular grid. A variational equation is obtained by forming the weighted residual of the time-discretised equation and imposing biperiodic boundary conditions. The test functions in the resulting integral are taken to be piecewise constant basis functions and the integrals are evaluated directly using a midpoint rule. Let ϕ be defined on an $M \times N$ grid in our computational domain, with N points in the x-direction and M points in the y-direction. We arrange the vector ϕ as a vector stacked by columns of ϕ's, i.e., $\phi = \begin{bmatrix} \phi_0 & \phi_1 & \dots & \phi_{N-1} \end{bmatrix}^T$, where $\phi_i = \begin{bmatrix} \phi_{i,0} & \phi_{i,1} & \dots & \phi_{i,M-1} \end{bmatrix}^T$. Thus, the discrete equations arising from (3.29) can be written in tensor-product form (see the Appendix) as follows [18]:

$$[P_N^{xx} \otimes P_M + P_N \otimes P_M^{yy} - \lambda_0 P_N \otimes P_M]\tilde{\phi}^{[k+1]} = R_H + \lambda_0 (P_N \otimes P_M)\bar{\phi}[ln(\frac{\phi}{\bar{\phi}}) - \frac{\tilde{\phi}}{\bar{\phi}}]^{[k]} \quad (3.30)$$

where P_N and P_M are the finite element mass matrices and P_N^{xx} and P_M^{yy} are the corresponding stiffness matrices. P_N and P_N^{xx} are of size $N \times N$ and P_M and P_M^{yy} are of size $M \times M$. When a uniform grid is used and periodic boundary conditions are assumed, then [18]

$$P_N = \frac{\Delta x}{4}\begin{bmatrix} 2 & 1 & & & 1 \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ 1 & & & 1 & 2 \end{bmatrix}, \quad P_M = \frac{\Delta y}{4}\begin{bmatrix} 2 & 1 & & & 1 \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ 1 & & & 1 & 2 \end{bmatrix},$$

$$P_N^{xx} = \frac{1}{\Delta x}\begin{bmatrix} -2 & 1 & & & 1 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{bmatrix}, \quad P_M^{yy} = \frac{1}{\Delta y}\begin{bmatrix} -2 & 1 & & & 1 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{bmatrix}.$$

The R_H term of the right side of the discrete Helmholtz equation (3.30) is given by

$$R_H = \frac{2}{\Delta t} R_D - \lambda_0 (P_N \otimes P_M) R_\phi \quad (3.31)$$

$$R_D = -\begin{bmatrix} D_x \otimes P_M & P_N \otimes D_y \end{bmatrix} \begin{bmatrix} a & b \\ -b & a \end{bmatrix} \begin{bmatrix} R_u \\ R_v \end{bmatrix}. \quad (3.32)$$

Here a and b are constants. The matrices D_x and D_y correspond to the partial derivatives $-\partial/\partial x$ and $-\partial/\partial y$ in finite element form once boundary conditions have been imposed.

3.4 The FFT Solver for the Helmholtz Equation

The Helmholtz equation to be solved at each step of iteration (3.30) is of the form

$$[P_N^{xx} \otimes P_M + P_N \otimes P_M^{yy} - \lambda_0 P_N \otimes P_M]\phi = H\phi = R \tag{3.33}$$

Figure 3.2 shows the almost block tridiagonal matrix structure of $P_N^{xx} \otimes P_M$, $P_N \otimes P_M^{yy}$, $P_N \otimes P_M$, and hence H as well.

It is known that the matrices P_N and P_N^{xx} can be simultaneously diagonalized by an orthogonal matrix W:

$$W^{-1} P_N^{xx} W = \Lambda_{xx}, \quad W^{-1} P_N W = \Lambda, \tag{3.34}$$

where W is an $N \times N$ matrix with $W_{jk} = w^{jk}$, $w = e^{2\pi i/N}$, $\Lambda_{xx} = \mathrm{diag}(p_0^{(2)}, p_1^{(2)}, \ldots, p_{N-1}^{(2)})$, $\Lambda = \mathrm{diag}(p_0, p_1, ..., p_{N-1})$ are diagonal matrices, and i is the imaginary unit.

The following transformation is carried out to solve (3.33).

$$(W^{-1} \otimes I_M)\phi = \hat{\phi} \tag{3.35}$$

$$(W^{-1} \otimes I_M)R = \hat{R} \tag{3.36}$$

$$(W^{-1} \otimes I_M)H(W \otimes I_M) = \hat{H} \tag{3.37}$$

The transformed system

$$[\Lambda_{xx} \otimes P_M + \Lambda \otimes P_M^{yy} - \lambda_0 \Lambda \otimes P_M]\hat{\phi} = \hat{R} \tag{3.38}$$

is block diagonal.

Figure 3.3 shows the block diagonal matrix structure of the coefficient matrix for system (3.38). Note that this system is decoupled into a set of almost tridiagonal systems which can be solved in parallel.

The solution of (3.33) is given by

$$\phi = (W \otimes I_M)[\Lambda_{xx} \otimes P_M + \Lambda \otimes P_M^{yy} - \lambda_0 \Lambda \otimes P_M]^{-1}(W^{-1} \otimes I_M)R \tag{3.39}$$

Thus, ϕ can be computed as follows.

1. Perform the Fourier transform

$$\hat{R} = (W^{-1} \otimes I_M)R. \tag{3.40}$$

2. Solve the decoupled tridiagonal systems

$$[\Lambda_{xx} \otimes P_M + \Lambda \otimes P_M^{yy} - \lambda_0 \Lambda \otimes P_M]\hat{\phi} = \hat{R} \tag{3.41}$$

to get $\hat{\phi}$.

3. Recover the solution ϕ by performing the inverse Fourier transform

$$\phi = (W \otimes I_M)\hat{\phi} \tag{3.42}$$

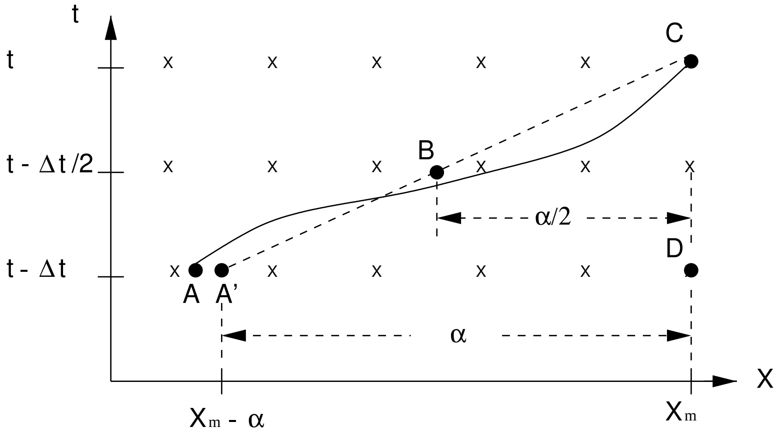

Figure 3.1: Schematic description of two-time-level advection. Actual (solid curve) and approximated (dashed line) trajectories that arrive at mesh point C at time $t + \Delta t$. Here α is the distance the particle is displaced in x in time Δt.

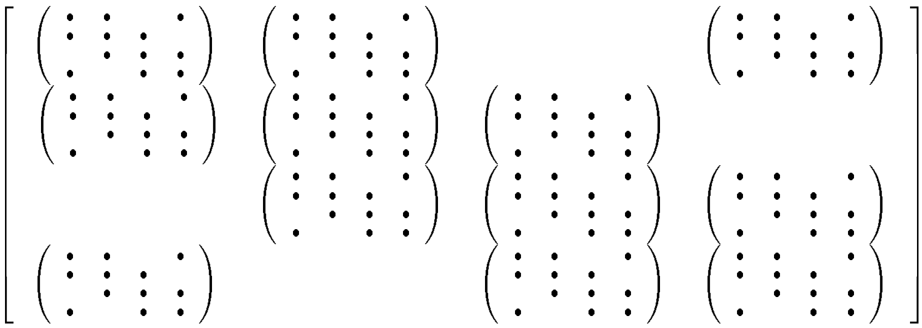

Figure 3.2: Structure of the almost block tridiagonal matrix that arises from the direct Helmholtz solver, with $M = N = 4$.

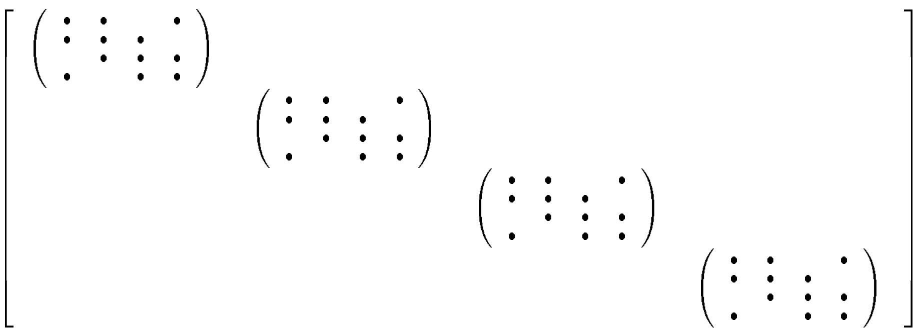

Figure 3.3: Structure of the block diagonal matrix that arises from the almost block tridiagonal matrix of Fig. 3.2 after transformation.

Chapter 4

A Parallel Solver for the 2D Shallow-Water Equations

4.1 Parallel Solution of the 2D Shallow-Water Equations

A data decomposition scheme is applied for the parallel solution of the shallow-water equations. In data decomposition, we keep the sequential formulation of the problem, but distribute the data and operations among the processors. The scalability of several data decomposition algorithms for finite difference atmospheric models has been analyzed by several authors [12, 18].

Several strategies exist within the data decomposition paradigm for dividing domains into sub-domains. Figure 4.1 shows a 1D and a 2D decomposition, two possible decompositions for a two-dimensional grid. In the 1D decomposition, the computational domain is decomposed in one direction only, while in the 2D decomposition, the computational domain is decomposed both in x and y coordinate directions.

In many cases, the computation is proportional to the volume of a sub-domain and the communication is proportional to the surface area. In such cases, a logical strategy is to partition the domain in a way that minimizes the surface area of each sub-domain relative to its volume. This keeps the computation-to-communication

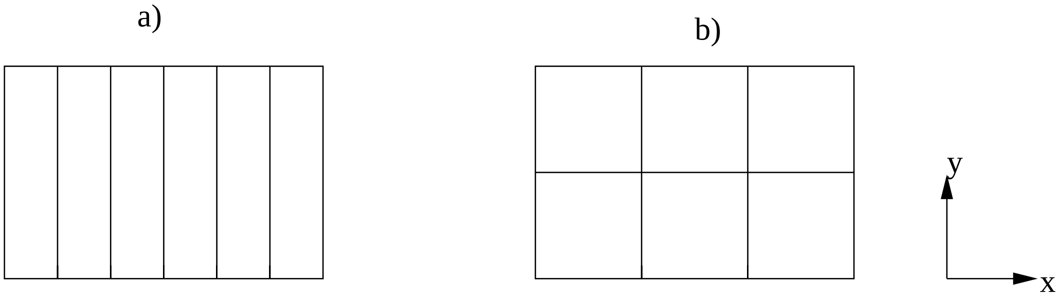

Figure 4.1: Data decomposition topologies for 1D(a) and 2D(b) decompositions

ratio high. In a 2D computational domain, if the number of grid points is $N \times N$, the communications at internal boundaries between sub-domains are of order $O(N)$ while the computations are of order at least $O(N^2)$. Obviously, the 2D decomposition has a smaller ratio between the boundary and the area than the 1D decomposition. In this thesis, the 2D decomposition is chosen in the parallel solver for the shallow-water equations.

The 2D decomposition involves assigning each sub-domain to a processor and solving the equations for that sub-domain on the respective processor. This partition scheme is favorable also due to the strong vertical dependencies in the physics parameterization process in the 3D atmospheric model. With the 2D decomposition, no global information is required at any particular grid point and interprocessor communications are required only at the boundaries of the sub-domains. The inner border of a sub-domain requires the outer border of the adjacent sub-domain during a time-step because of the spatial discretization. (See Figure 4.2).

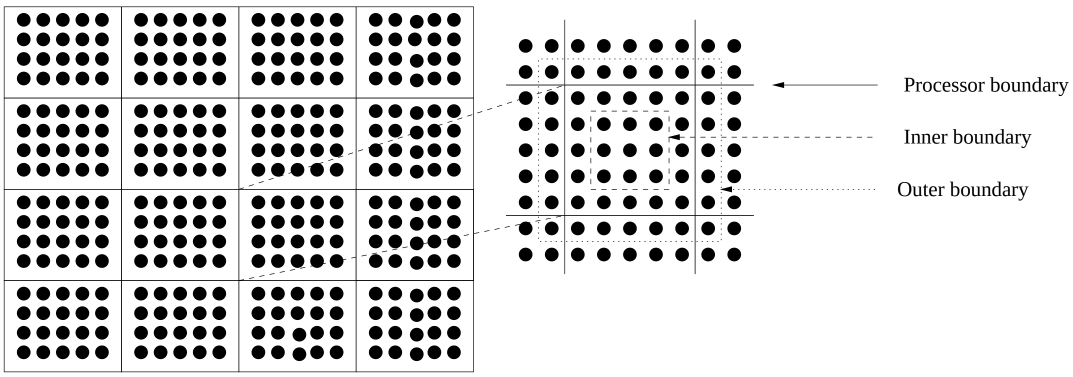

Figure 4.2: Spatial domain decomposition of the parallel shallow-water model.

At the start of an integration, the forecast domain is divided as evenly as possible into rectangular sub-domains which are distributed across the available processors. (See Figure 4.2). Let N and M be the number of grid points in the x and y-direction of the forecast domain. The total number of processors is $P = P_x \times P_y$, where P_x and P_y are the numbers of processors assigned to the x and y directions respectively. The number of grid points in each sub-domain is $\frac{N}{P_x} \cdot \frac{M}{P_y}$. For simplicity, we will assume that $\frac{N}{P_x}$ and $\frac{M}{P_y}$ are integers. Otherwise some domains will have one more row and/or column of grid points than the others. This will not change the asymptotic behavior of the algorithm. Each processor runs a copy of the numerical model defined on one of the sub-domains. The computation of each component of the right-hand sides R_u, R_v and R_ϕ at time $t - \Delta t$ in the equations (3.18)—(3.20) is carried out on the respective processor. Since ϕ_x and ϕ_y at grid point (i, j) is calculated as

$$(\phi_x)_{i,j} = \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x_i} \tag{4.1}$$

$$(\phi_y)_{i,j} = \frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta y_i} \tag{4.2}$$

a one grid-point overlap of ϕ values on neighboring sub-domains is sufficient to update the discrete right-hand sides R_u, R_v and R_ϕ according to (3.21)—(3.23).

Let $C = |u_m|\Delta t/\Delta x$ denote the *Courant Number*, where u_m is the maximum wind speed, Δt is the time step-size, and Δx is the space step-size. The Courant number has important implications for distributed memory implementations of both Eulerian and semi-Lagrangian advection schemes which employ data distribution of the computational grid. In an Eulerian leapfrog advection scheme, values of the advected field F are taken from a stencil containing the four nearest grid-points. Thus, a one grid-point wide overlap region is maintained on each processor and an exchange of values at these grid-points is required after each time step. The maximum length of a time step is limited by the Courant-Fredrichs-Lewy condition $C \leq 1$ for numerical stability. Further, the semi-Lagrangian treatment of advection requires the upstream interpolation in space. The parallel implementation of the traditional sequential algorithm requires a two grid-point wide overlap region between adjacent sub-domains to be maintained, when $C < 1$ to ensure the data surrounding a departure point is available. These communication patterns are all local neighbor-to-neighbor communications. Once the time discretization is handled and the right side of the Helmholtz equation is computed, the next step is the efficient solution of the Helmholtz equation. In the next section, we describe efficient parallel solvers for the Helmholtz equation.

4.2 Parallel Solution of the Helmholtz Problem

In solving the Helmholtz equation using the FFT direct method, a processor must have access to all the data in a row to perform the FFTs (Eqns. (3.40) and (3.42)). To carry out the tridiagonal solve (Eqn. (3.41)), a processor must be able to access all the data in a column.

Assume that we start with a 2D x-y decomposition with $P_x \times P_y$ processors (phase 1). We then redistribute the data into a 1D x-direction decomposition (phase 2), where rows of grid points in the x-direction are distributed among the processors, so that the FFTs (Eqn. (3.40)) can be carried out independently. Then the data are redistributed into a 1D y-direction decomposition (phase 3), where columns of grid-points in the y-direction are distributed among the processors and the tridiagonal solves can be carried out on all processors independently. The inverse FFTs (3.42) can be carried out independently after another data transposition from phase 3 to phase 2.

The parallel algorithm for the direct Helmholtz solver is the following.

```
begin
  1) Redistribute from the 2D x-y block partition to
     a 1D x-direction block partition;
  2) Carry out the FFT (Eqn.(3.40)) in the x-direction;
  3) Transpose from the 1D x-direction block partition
     to a 1D y-direction block partition;
  4) Carry out the Gaussian elimination in the y-direction (Eqn.(3.41));
```

```
  5) Transpose from the y-direction partition to a x-direction partition;
  6) Carry out the inverse FFT (Eqn. (3.42)) in the x-direction;
  7) Redistribute from the 1D x-direction block partition
     to a 2D x-y partition;
end
```

This implies that a global data transposition is required in steps 3 and 5 and a more local data redistribution in steps 1 and 7.

The advantage of this approach is that it keeps the main numerical computations of the sequential algorithm unchanged, so that they can be carried out independently on each processor.

However the global data transposition requires a *complete exchange* communication operation among all processors. That is, it requires each processor to send data to every other processor. This is one of the most demanding operations in terms of bandwidth and communication latency for distributed systems. The data transposition problem also arises in various other applications, such as the Alternating Direction(ADI) Method and spectral models.

4.3 Data Layouts for the Helmholtz Solver

Assume the shallow-water equations are to be solved on a rectangle domain, discretized into $N \times M$ grid-points, where N is the number of grid-points in the x-direction and M is the number of grid-points in the y-direction. P processors are used to solve this problem. To carry out FFTs, contiguous blocks of rows of the data are distributed among the P processors, as evenly as possible to keep the load balanced. This type of data distribution is noted as phase 2. We denote the processors as $p_i, i = 0, \ldots, P-1$. Processor p_i holds m_i rows of the data. To keep the load balanced, the following algorithm is used to calculate m_i.

```
m = [M/P];
if (i <= (M - P * m))
   m_{i} = m + 1;
```

```
else
   m_{i} = m;
```

Let $r_i = m_0 + \ldots + m_{i-1}$. Processor p_i holds all the data from row r_i to row r_{i+1}.

A similar algorithm is used to distribute columns of data among the processors at phase 3. Processor p_i holds n_i columns of data, where n_i is computed as follows.

```
n = [N/P];
if (i <= (N - P * N))
   n_{i} = n + 1;
else
   n_{i} = n;
```

Let $c_i = n_0 + \ldots + n_{i-1}$. Processor p_i holds all the data from column c_i to column c_{i+1}.

In phase 2, the grid points on processor p consist of small data blocks $blk_{p,0}, \ldots, blk_{p,(P-1)}$, where blk_{pq} contains grid-points (i, j), where $c_p \leq i < c_{p+1}, r_q \leq j < r_{q+1}$. The grid points on processor p consist of small blocks $blk_{0,p}, \ldots, blk_{(P-1),p}$ in phase 3.

4.4 Algorithms for the Global Data Transposition Problem

To redistribute the data from phase 2 to phase 3, processor p must transfer blk_{pq} to processor q and get blk_{qp} from processor q ($q = 0, \ldots, (P-1)$, $q \neq p$).

Figure 4.3 illustrates how complete exchange arises when transposing the data on a 4-processor system. Arrows between blocks identify exchange-pairs involving processor p_0. Note the p_0 must send and receive, in some way, one block of data to/from each processor. The operation is costly; it is therefore worthwhile to implement it efficiently.

The *Direct Exchange* and *Ring Exchange* schemes are two well-known complete exchange communication schemes. They are shown in Figure 4.4. Both the Direct

Exchange and Ring Exchange algorithms requires $O(P)$ messages for every processor. As we know, $t_s >> t_w$ for most modern parallel computers. If P is large (with respect to the problem size), the startup time of the communication may dominate the communication time. To reduce the number of messages sent by every processor, we extend the Ring Exchange algorithm to the Mesh Exchange and Cube Exchange algorithms to take advantage of the high dimensional interprocessor communication capability. In the following, the details of the Direct, Ring, Mesh and Cube Exchange algorithms are described. For all exchange schemes, we show the code run by processor indexed 'me'.

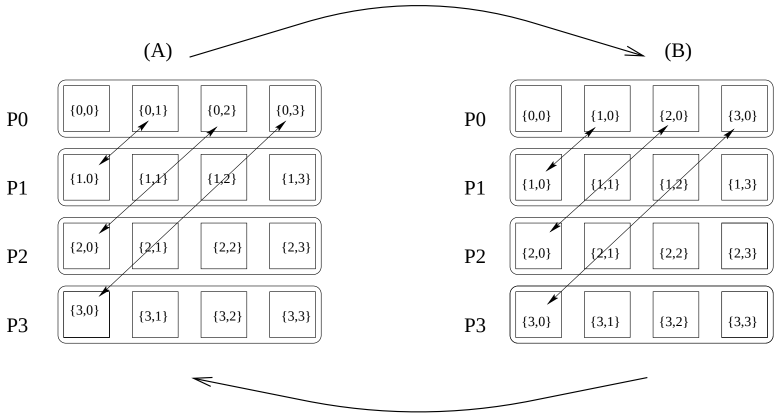

Figure 4.3: The exchange of data that processor P_0 needs to perform for the global data transposition problem on a 4-processor system.

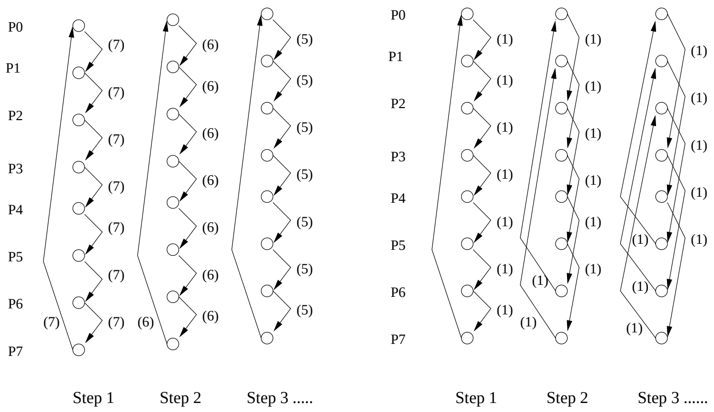

Figure 4.4: Direct Exchange and Ring Exchange schemes on 8-processor system. For brevity, only the first 3 of the 7 steps required are shown for each scheme. The number in parentheses denotes the amount of data to be transmitted.

The Direct Exchange Algorithm

The first algorithm is called Direct Exchange, and is quite straightforward. Every processor exchanges messages with the other $P-1$ processors directly. The message size is $m_1 = \frac{M \times N}{P^2}$.

```
DirectExchange(){
        for i = 1 to (P-1) do{
            precv = (me+i) mod P
            send msg_send(me,precv) to processor precv;
        }
        for i = 1 to (P-1) do{
            psend = (me+i) mod P
            receive msg_recv(psend,me) from processor psend;
        }
}
```

where $msg\_send(me, precv)$ is the data block $blk_{me,precv}$ and $msg\_recv(psend, me)$ is the data block $blk_{psend,me}$, for processor me, as described in the previous section.

If there is a direct link from each processor to each other, using the message passing cost model from Chapter 2, the total communication cost for each processor is

$$T_1 = P \cdot (t_s + t_w \cdot \frac{M \times N}{P^2}). \tag{4.3}$$

In practice, however, the parallel machine may not have sufficient links to guarantee that every processor can exchange messages with all others concurrently without contention. For a workstation cluster connected by an ethernet, P processors share one communication channel. At each step, P processors try to send messages at the same time on one communication channel. According to Foster [4], the communication time can be approximated by

$$\begin{aligned} T_1 &= P \cdot (t_s + t_w \cdot \frac{M \times N}{P^2} \cdot P) \\ &= P \cdot t_s + t_w \cdot (M \times N) \end{aligned} \tag{4.4}$$

The Ring Exchange Algorithm

A linear array of processors with a wraparound connection is referred to as a *ring*. Figure 4.5 shows a ring of four processors and the data flow of the Ring Exchange algorithm. To communicate between processors, a message is passed to the next (or the previous) processor repeatedly until it reaches its destination. The processors are indexed by a processor number i, where $i = 0, \ldots, (P-1)$. Let $blk(i,j)$ represents the data block to be sent from processor i to processor j. The size of the data block is $m_1 = \frac{MN}{P^2}$. Since each processor communicates only with its neighbors, a store and forward technique must be used to route the data to its destination. Between each step synchronizations are required to make sure that the data has been received before trying to forward it to the next processor. To reduce the total number of messages, the data blocks to be sent are packed into one packet whenever possible.

```
RingExchange(){
  for i = 1 to P-1 do{
    send msg_send(i) to next_processor;
    receive msg_recv(i) from prev_processor;
    extract the data block blk(mod(me-i,P),me) from the received message;
  }
}
```

At each step i, $msg\_send(i)$ consists of $(P-i)$ data blocks $\{blk(me \ominus (i-1), j) : j = me \oplus 1, \ldots, me \oplus (P-i)\}$ of size m each. The operators $\ominus$ and $\oplus$ denote the '-' and '+' operators modulo P. For example, processor # $(me \ominus i) = mod(me - i, P)$ means the processor that is i links away from me in one direction of the *ring*, and processor # $(me \oplus i) = mod(me + i, P)$ indicates the processor that is i links away from me in the other direction. Similarly, the received message $msg\_recv(i) = \{blk(me \ominus i, j) : j = me, \ldots, me \oplus (P - i - 1)\}$. Figure 4.5 illustrates the steps of the Ring Exchange algorithm on a 4-processor ring.

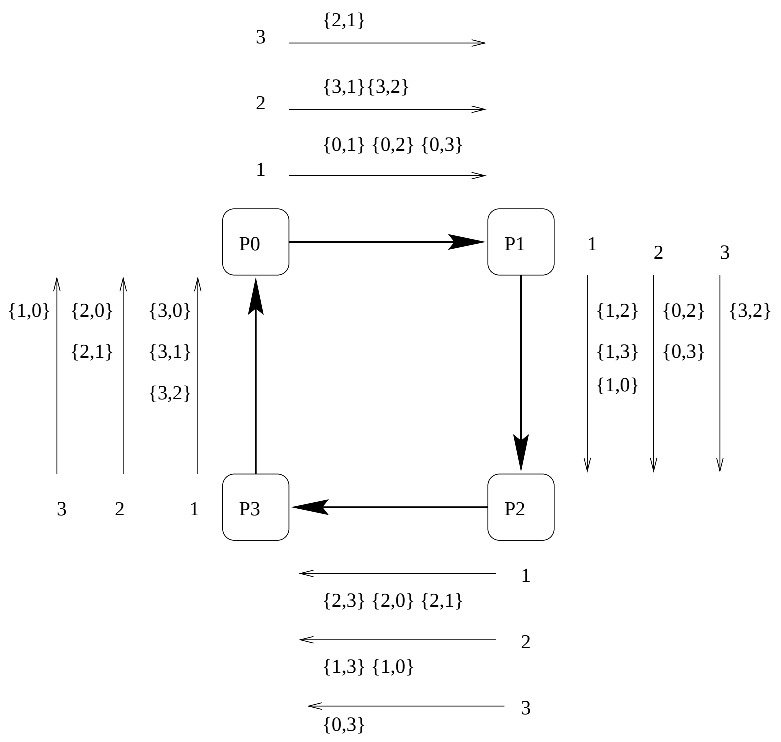

Figure 4.5: Data Communications on a 4-processor Ring. The label of each message is of the form $\{i, j\}$, where i is the label of the processor from which the message originally comes, and j is the label of the processor that is the final destination of the message. Note that the messages are formed by concatenating several individual data blocks.

The communication cost of this algorithm for each processor is

$$T_{ring} = \sum_{i=1}^{P-1}(t_s + t_w \cdot (P-i) \cdot m_1) = (P-1)(t_s + t_w \cdot \frac{P}{2} \cdot m_1). \tag{4.5}$$

Note that $P^2 \cdot m_1 = M \cdot N$. Thus (4.5) can be written as

$$T_{ring} = (P-1)(t_s + t_w \cdot \frac{M \cdot N}{2 \cdot P}). \tag{4.6}$$

The Mesh Exchange Algorithm

The *Ring Exchange* Algorithm can be extended to a *Mesh Exchange* when a mesh interprocessor connection is available. The *Mesh Exchange* algorithm is based on applying the Ring Exchange algorithm in each of the two dimensions, treating rows and columns of the processor mesh as rings. Let P_x and P_y be the number of processors in x and y directions respectively. The total number of processors is $P = P_x \times P_y$. The processors are indexed by $p_{i,j}$, where $i = 0, 1, \ldots, (P_x - 1)$, and $j = 0, 1, \ldots, (P_y - 1)$ represents the processor index in the x and y-directions. Figures 4.6 and 4.7 show the 16 processors with their mesh connection. Given a fixed j, processors in the same row $\{p_{i,j}, i = 0, \ldots, P_x\}$ form a ring in the x-direction. On the other hand, given a fixed i, processors in the same column $\{p_{i,j}, j = 0, \ldots, P_y\}$ form a ring in the y-direction.

The transposition can be completed by exchanging $P_x + P_y - 2$ messages for every processor. The algorithm proceeds in two phases. Given a processor $p_{i0,j0}$, a complete data exchange using the Ring Exchange algorithm is carried out in the x-direction. After this data exchange, processor $p_{i,j0}$, $i = 0, \ldots, P_x$, will have all the data blocks needed for the processors in column j_0. The size of the data block to be exchanged between any pairs of the processors within the ring is then $P_y \cdot m_1$. Obviously it requires $P_x - 1$ messages for every processor. By applying the communication model (4.5) for the Ring Exchange algorithm, we can approximate the communication time involving the processors in the x-direction by

$$
\begin{aligned}
T_x &= (P_x - 1) * (t_s + t_w \cdot \frac{P_x}{2} \cdot P_y \cdot m_1) \\
&= (P_x - 1) * (t_s + t_w \cdot \frac{P_x}{2} \cdot P_y \cdot \frac{MN}{P^2}) \\
&= (P_x - 1) * (t_s + t_w \cdot \frac{MN}{2P}).
\end{aligned}
$$

Since no communication is required between the processors in the y-direction in the first phase, the Ring Exchange operations can be carried out independently. Therefore, the overall communication cost of the first phase is also equivalent to T_x.

Now all the data required by processor $p_{i0,j0}$ has been received by processors $\{p_{i,j0}, i = 0, \ldots, P_x\}$. Processor $p_{i0,j0}$ will receive all the required date after a complete exchange using the Ring Exchange algorithm in the y-direction (See Figure 4.7). The number of messages is P_y and the size of each data block to be exchanged between any pairs in the ring is $P_x \cdot m_1$. The communication cost of this phase is

$$\begin{aligned} T_y &= (P_y - 1) * (t_s + t_w \cdot \frac{P_y}{2} \cdot P_x \cdot m_1) \\ &= (P_y - 1) * (t_s + t_w \cdot \frac{P_y}{2} \cdot P_x \cdot \frac{MN}{P^2}) \\ &= (P_y - 1) * (t_s + t_w \cdot \frac{MN}{2P}). \end{aligned}$$

This completes the transposition operation.

The total communication time for each processor is

$$\begin{aligned} T_{mesh} &= T_x + T_y \\ &= (P_x + P_y - 2) \cdot (t_s + t_w \cdot \frac{MN}{2P}). \end{aligned} \tag{4.7}$$

If $P_x = P_y = \sqrt{P}$, this becomes

$$T_{mesh} = 2 \cdot (\sqrt{P} - 1) \cdot (t_s + t_w \cdot \frac{MN}{2 \cdot P}). \tag{4.8}$$

In total, every processor needs to send and receive $O(P^{1/2})$ messages only. The following figures illustrate this algorithm.

```
MeshExchange(){
    perform the Ring Exchange algorithm in the x-direction;
    perform the Ring Exchange algorithm in the y-direction;
}
```

The Cube Exchange Algorithm

When a 3D torus connection is available, the algorithm can be improved further. The *Cube Exchange* algorithm applies the Ring Exchange algorithm to the *x, y* and *z* directions to accomplish the complete exchange on a 3D torus network topology.

Let P_x, P_y and P_z be the number of processors in the x, y and z-directions. The total number of processors is $P = P_x \times P_y \times P_z$. The processors are indexed by

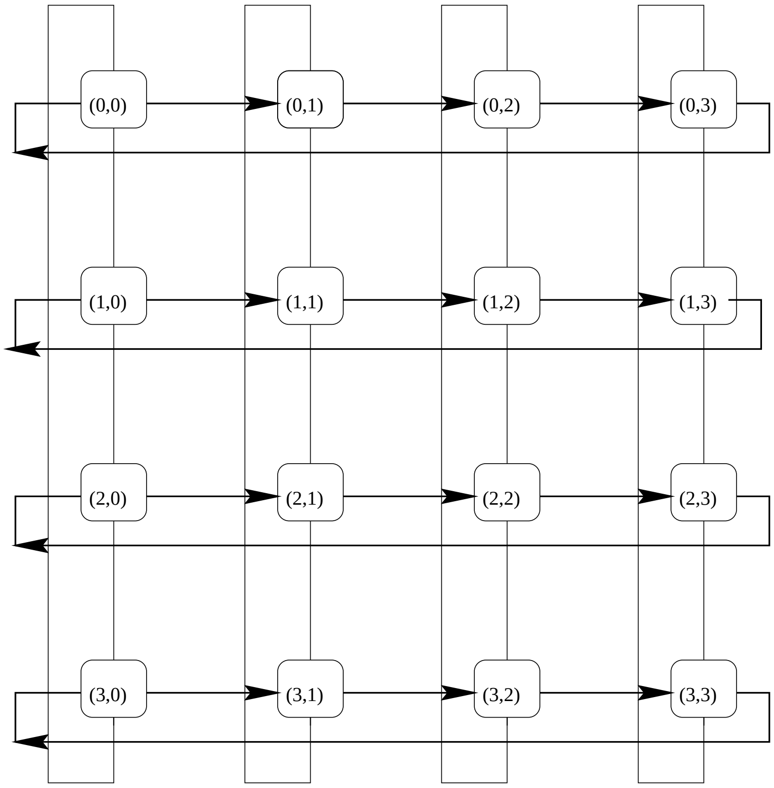

Figure 4.6: Mesh Exchange algorithm phase 1. The processors perform the Ring Exchange algorithm along the rings in the x-direction simultaneously.

$p_{i,j,k}$, where $i = 0, \ldots, (P_x - 1)$, $j = 0, \ldots, (P_y - 1)$, $k = 0, \ldots, (P_z - 1)$. Each node is directly connected to 6 processors in the x, y and z-directions. Processors $p_{i,j0,k0}$ where $i = 0, \ldots, (P_x - 1)$ form a *ring* in the x-direction for a given j_0 and k_0. Processors $p_{i0,j,k0}$ where $j = 0, \ldots, (P_y - 1)$ form a *ring* in the y-direction for a given i_0 and k_0. Processors $p_{i0,j0,k}$ where $k = 0, \ldots, (P_z - 1)$ form a *ring* in the z-direction for a given i_0 and j_0. Processor $p_{i0,j0,k0}$ belongs to all three rings.

The Cube Exchange algorithm proceeds in three phases. The first phase applies the Ring Exchange algorithm to the processors in the x-direction, and requires P_x messages. The size of the data block to be exchanged between any pair of processors in this phase is $P_y \cdot P_z \cdot m_1$. The communication cost for every processor in phase 1 is

$$\begin{aligned} T_x &= (P_x - 1) \cdot (t_s + t_w \cdot \frac{P_x}{2} \cdot P_y \cdot P_z \cdot m_1) \\ &= (P_x - 1) \cdot (t_s + t_w \cdot \frac{P_x}{2} \cdot P_y \cdot P_z \cdot \frac{MN}{P^2}) \end{aligned}$$

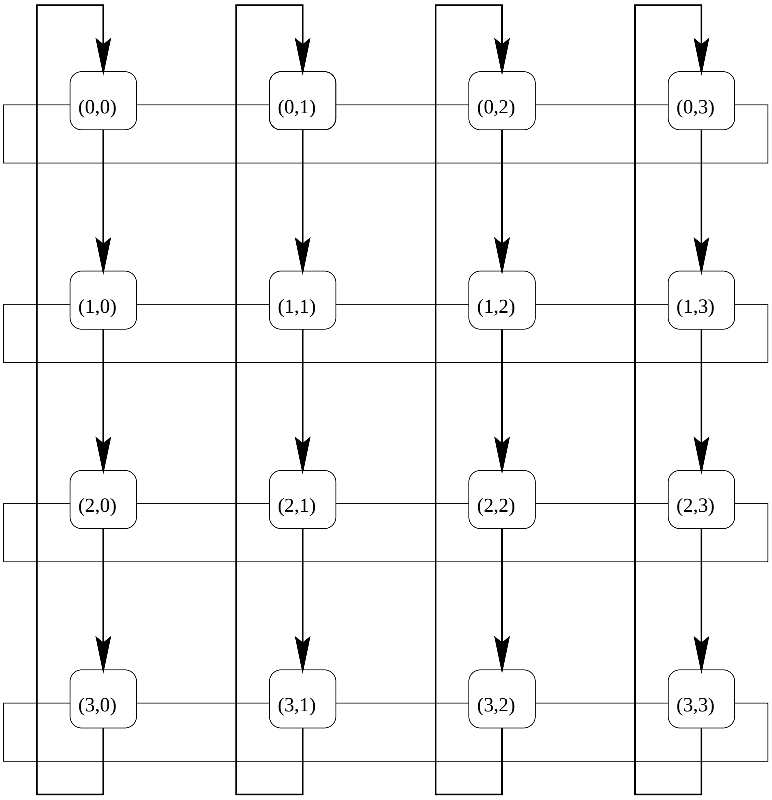

Figure 4.7: Mesh Exchange algorithm phase 2. The processors perform the Ring Exchange algorithm along the rings in the y-direction simultaneously.

$$= (P_x - 1) \cdot (t_s + t_w \cdot \frac{MN}{2P}). \tag{4.9}$$

Similarly, in the second and third phases, the *Ring Exchange* algorithm is applied to the rings in the y and z-directions, respectively. The communication costs for the Ring Exchange in the y and z-direction are $T_y = (P_y - 1) \cdot (t_s + t_w \cdot \frac{MN}{2P})$, and $T_z = (P_z - 1) \cdot (t_s + t_w \cdot \frac{MN}{2P})$ respectively.

The total communication cost for the three phases is

$$\begin{aligned} T_{cube} &= T_x + T_y + T_z \\ &= (P_x + P_y + P_z - 3) \cdot (t_s + t_w \cdot \frac{MN}{2P}). \end{aligned} \tag{4.10}$$

If $P_x = P_y = P_z = \sqrt[3]{P}$, this becomes

$$T_{cube} = 3 \cdot (\sqrt[3]{P} - 1) \cdot (t_s + t_w \cdot \frac{MN}{2 \cdot P}) \tag{4.11}$$

The total number of messages required by each processor is $(P_x + P_y + P_z - 3)$, which is of order $O(\sqrt[3]{P})$. The following figures illustrate the Cube Exchange algorithm.

```
CubeExchange(){
   perform the Ring Exchange algorithm in the x-direction;
   perform the Ring Exchange algorithm in the y-direction;
   perform the Ring Exchange algorithm in the z-direction;
}
```

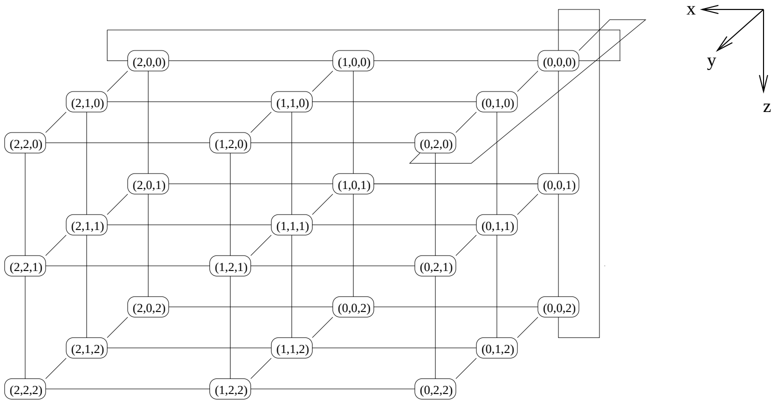

Figure 4.8: 3D torus connections. For clarity, some of the links are not shown. There are 6 links for each processor connecting the 6 neighbor processors in 3 directions.

Figures 4.9–4.11 illustrate the message flows in the three phases of the Cube Exchange algorithm.

4.5 Parallel Complexity and Scalability Analysis

4.5.1 Communication and Storage

Again we assume that $P = P_x \times P_y$ processors are used to solve the shallow-water equations, and each sub-domain has $m \times n$ grid-points, where $m = \frac{M}{P_y}$ and $n = \frac{N}{P_x}$.

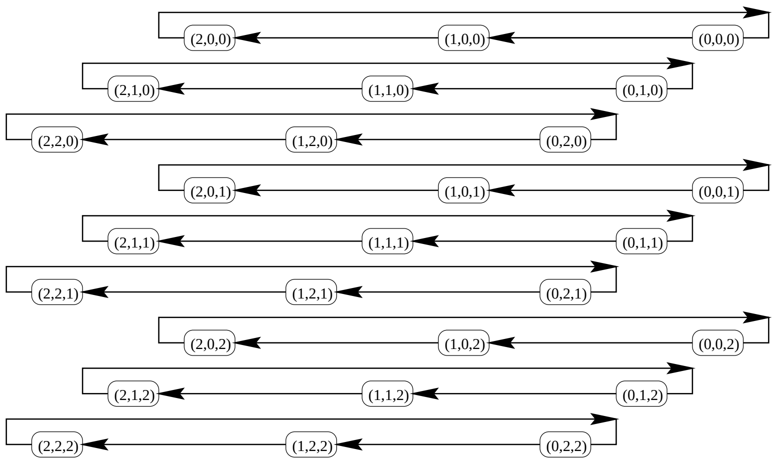

Figure 4.9: Cube Exchange algorithm phase 1. The processors perform the Ring Exchange algorithm along the rings on y-z plane simultaneously.

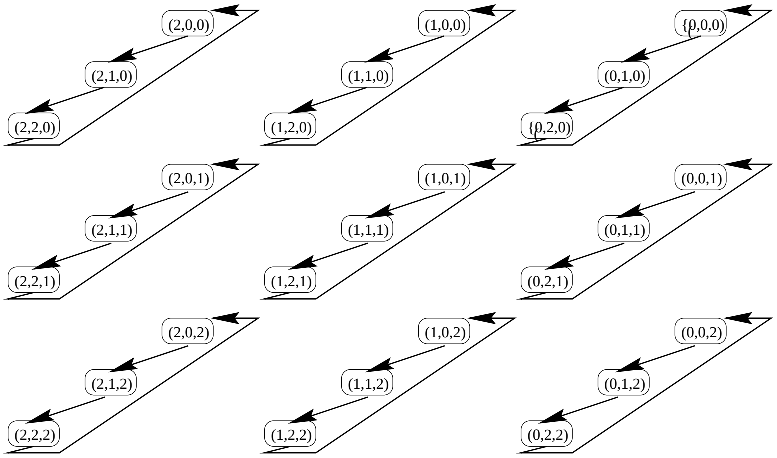

Figure 4.10: Cube Exchange algorithm phase 2. The processors perform the Ring Exchange algorithm along the rings on x-z plane simultaneously.

Tables 4.1 and 4.2 summarize the communications required in addition to the global data transposition for the parallel shallow-water model. In the parts of the program covered by these tables, it is assumed that all data (different variables) to be sent in one direction can be sent in one message.

Table 4.1 shows the nearest neighbor communication involved in the parallel shallow-water model, which includes communication overlap regions for advection. The overlap regions include 3 boundary grid-points of the u, v and the right side of equations (3.18)–(3.20). The size of the messages is $15 \cdot \frac{M}{P_y} = 15m$ between East–West neighbors, and $15 \cdot \frac{N}{P_x} = 15n$ between the North–South neighbors. It takes 4

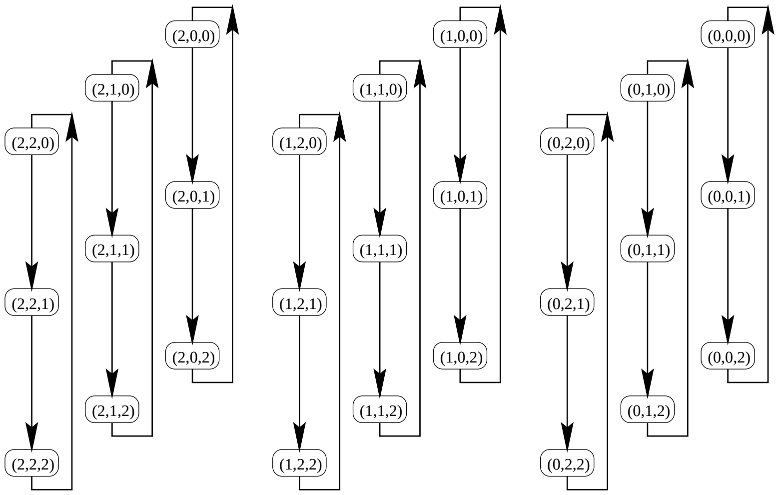

Figure 4.11: Cube Exchange algorithm phase 3. The processors perform the Ring Exchange algorithm along the rings on x-y plane simultaneously.

	North	South	West	East
Advection	$15 \cdot n$	$15 \cdot n$	$15 \cdot m$	$15 \cdot m$
Prep. Helmholtz	$3 \cdot n$	$3 \cdot n$	$3 \cdot m$	$3 \cdot m$
Update variables	$2 \cdot n$	$2 \cdot n$	$2 \cdot m$	$2 \cdot m$

Table 4.1: Communication of floating-point numbers between neighboring domains per time step.

messages for every processor to exchange these data with its 4 neighbor processors. Preparing the right sides of the Helmholtz equation requires 4 messages. The size of the messages is $3\frac{M}{P_y} = 3m$ between East–West neighbors, and $3\frac{N}{P_x} = 3n$ between North–South neighbors.

Updating ϕ after the Helmholtz solver requires 4 messages between neighboring processors. The size of the messages is $2\frac{M}{P_y} = 2m$ between East–West neighbors, and $2\frac{N}{P_x} = 2n$ between North–South neighbors.

Data redistribution between phase 1 (2D partition) and phase 2 (1D partition) for the Helmholtz FFT direct solver involves P_x messages, each of size $\frac{N}{P_x^2} \cdot \frac{M}{P_y} = \frac{mn}{P_x}$.

Phase 1 $\rightarrow$ Phase 2		Phase 2 $\rightarrow$ Phase 1	
Number of messages	Size of message	Number of messages	Size of message
P_x	$\frac{m \cdot n}{P_x}$	P_x	$\frac{m \cdot n}{P_x}$

Table 4.2: Communication of floating-point numbers between phase 1 and phase 2 of the direct Helmholtz solver.

The total neighbor-to-neighbor communication time is

$$\begin{aligned} T_{neighbor-to-neighbor} &= 2(t_s + 15\frac{M}{P_y}) + 2(t_s + 15\frac{N}{P_x}) \\ &+ 2(t_s + 3\frac{M}{P_y}) + 2(t_s + 3\frac{N}{P_x}) \\ &+ 2(t_s + 2\frac{M}{P_y}) + 2(t_s + 2\frac{N}{P_x}) \\ &= 12t_s + 40(\frac{M}{P_y} + \frac{N}{P_N}) \end{aligned}$$

Table 4.2 shows the communication of the data redistribution from the 2D x-y block partition (phase 1) to 1D x-direction (phase 2) partition and the reverse process in the direct Helmholtz solver. Assume 2 iterations are used in the fixed-point method for the non-linear Helmholtz problem, then the total communication cost for the parallel shallow-water model can be approximated by

$$\begin{aligned} T_{comm} &= T_{neighbor-to-neighbor} + T_{comm12} + T_{comm23} \\ &= (12t_s + 40t_w(\frac{N}{P_x} + \frac{M}{P_y})) + 2 \cdot P_x(t_s + t_w \cdot \frac{M \cdot N}{P \cdot P_x}) + T_{comm23}, \end{aligned} \quad (4.12)$$

where T_{comm23} is the communication cost of the global transposition between phase 2 and phase 3.

We now consider the total amount of memory used on each processor. The amount of storage is dominated by the variables u, v and ϕ. We need to store the wind at time levels $t - 2dt$, $t - dt$, $t - dt/2$ and t, ϕ and the divergence D at time level t, a work array for ϕ at the Helmholtz iteration, $log(1 + \frac{\phi}{\bar{\phi}})$, ϕ_x and ϕ_y. We also need to store the displacement α of the departure points. We can then approximate the memory usage per processor measured in floating-point numbers by

$$Mem = 16(\frac{N}{P_x} + 2)(\frac{M}{P_y} + 2). \quad (4.13)$$

Description of the computations	Number of floating-point operations
Compute the upstream departure points	$80 \cdot M \cdot N$
Compute right-hand sides R_u, R_v and R_ϕ	$13 \cdot M \cdot N$
Interpolate RHS's at time level $t - \Delta t$	$190 \cdot M \cdot N$
Compute RHS of Helmholtz problem	$20 \cdot M \cdot N$
Solve the non-linear Helmholtz problem	$44 \cdot M \cdot N + 5 \cdot M \cdot N \log N$
Update variables and others	$52 \cdot M \cdot N$

Table 4.3: Computational complexity in terms of the numbers of floating-point operations for the different components of the semi-implicit semi-Lagrangian model per time step.

4.5.2 Scalability Analysis

The shallow-water model is solved on a rectangular domain with $M \times N$ grid-points. The computational complexity of each time step of the sequential shallow-water model is estimated as $T_1 \simeq 400 \cdot M \cdot N + 5 \cdot M \cdot N \cdot \log N = O(MN \log N)$ floating-point operations. Table 4.3 shows the breakdowns of the different components of the semi-implicit semi-Lagrange shallow-water model.

As we have noticed, the numerical computations of the shallow-water model are perfectly parallelized. No extra computation is required for the parallel algorithm. The idle time due to the load imbalance and the time spent on duplicate computation for the overlap regions does not affect the asymptotic behavior of the computational complexity.

Therefore, the computing time for one time-step of the shallow-water model on $P = P_x \times P_y$ processors can be approximated by $T_{comp} = T_1/P$, and the elapsed time per time step for the shallow-water model can be approximated by

$$\begin{aligned} T_P &= T_{comp} + T_{comm} \\ &= T_{comp} + T_{neighbor-to-neighbor} + T_{comm12} + T_{comm23} \\ &= \frac{T_1}{P} + (12t_s + 40t_w(\frac{N}{P_x} + \frac{M}{P_y})) + 2 \cdot P_x(t_s + t_w \cdot \frac{M \cdot N}{P \cdot P_x}) + T_{comm23}. \end{aligned}$$

Assuming $P_y = \nu P_x$ and $M = \lambda N$, we have

$$T_P = \frac{T_1}{P} + 12t_s + 40t_w\sqrt{\nu}(1+\frac{\lambda}{\nu})\frac{N}{\sqrt{P}} + \frac{2}{\sqrt{\nu}}\sqrt{P}t_s + 2t_w\frac{MN}{P} + T_{comm23}. \qquad (4.14)$$

When the Ring Exchange algorithm is used, the global transposition time can be approximated by

$$\begin{aligned} T_{comm23} &= 2T_{ring} \\ &= 2P(t_s + t_w \cdot \frac{MN}{2P}) \\ &= 2P(t_s + t_w\lambda\frac{N^2}{2P}) \\ &= 2Pt_s + t_w\lambda N^2. \end{aligned} \qquad (4.15)$$

In this case the elapsed time is

$$\begin{aligned} T_P &= \frac{T_1}{P} \\ &+ 12t_s + 40t_w\sqrt{\nu}(1+\frac{\lambda}{\nu})\frac{N}{\sqrt{P}} \\ &+ \frac{2}{\sqrt{\nu}}\sqrt{P}t_s + 2\lambda t_w\frac{N^2}{P} \\ &+ 2Pt_s + t_w\lambda N^2. \end{aligned} \qquad (4.16)$$

Using the definition in Chapter 2, the problem size of this parallel system is

$$\begin{aligned} W &= T_1 \\ &= c_1\lambda N^2 + c_2\lambda N^2 \log N. \end{aligned} \qquad (4.17)$$

The overhead function is

$$\begin{aligned} T_o(W,P) &= PT_P - T_1 \\ &= 12t_sP + 40t_w\sqrt{\nu}(1+\frac{\lambda}{\nu})N\sqrt{P} \\ &+ \frac{2}{\sqrt{\nu}}P^{3/2}t_s + 2\lambda t_w N^2 \end{aligned}$$

$$+ \quad 2t_s P^2 + t_w \cdot PN^2. \tag{4.18}$$

According to Chapter 2, the ratio $T_o(W,P)/W$ must be kept under a constant value to maintain parallel efficiency. This implies each of the terms in (4.18) must be kept under the same order of W, i.e.,

$N^2 \log N \propto P \Rightarrow N \propto P^{1/2}$ will be sufficient.

$N^2 \log N \propto NP^{1/2} \Rightarrow N \propto P^{1/2}$ will be sufficient.

$N^2 \log N \propto P^{3/2} \Rightarrow N \propto P^{3/4}$ will be sufficient.

$N^2 \log N \propto N^2 \Rightarrow$ OK.

$N^2 \log N \propto P^2 \Rightarrow N \propto P$ will be sufficient.

$N^2 \log N \propto N^2 P \Rightarrow N \propto 2^P$.

The last relation shows that the growth rate of N needed to maintain the parallel efficiency is determined by the global transposition communication time.

Thus, when the Ring Exchange algorithm is used, the isoefficiency function is

$$f_{ring}(P) = KT_o = K(N^2 \log N) = K(P2^{2P}), \tag{4.19}$$

where K is constant and $N = O(2^P)$. That is, the number N of grid points in one dimension must grow at the order of $O(2^P)$. According to (4.13), the memory required by every processor to maintain efficiency must grow at the order of $O(\frac{2^{2P}}{P})$.

When the Mesh Exchange algorithm is used for the global data transposition, we have

$$\begin{aligned} T_{comm23} &= T_{mesh} \\ &= 2P^{1/2}(t_s + t_w\lambda \cdot \frac{N^2}{2P}) \\ &= 2t_s P^{1/2} + \frac{t_w\lambda}{2}\frac{N^2}{P^{1/2}}. \end{aligned} \tag{4.20}$$

Proportionality between the problem size W and the global transposition time

part of the overhead function gives
$N^2 \log N \propto P^{3/2} \Rightarrow N \propto P^{3/4}$ will be sufficient.
$N^2 \log N \propto P^{3/2} \Rightarrow N \propto P^{3/4}$ will be sufficient.
$N^2 \log N \propto N^2 P^{1/2} \Rightarrow N \propto 2^{P^{1/2}}$.

The isoefficiency function is $f_{mesh}(P) = O(P^{1/2} 2^{2P^{1/2}})$, and the memory required by every processor to maintain the efficiency must grow at the order of $O(\frac{2^{2P^{1/2}}}{P})$.

Similarly, when the Cube Exchange algorithm is used,

$$\begin{aligned} T_{comm23} &= T_{cube} \\ &= 3P^{1/3}(t_s + t_w \cdot \frac{MN}{2P}) \\ &= 3t_s P^{1/3} + \frac{t_w \lambda}{2} \cdot N^2 P^{2/3}. \end{aligned} \tag{4.21}$$

Proportionality between W and the global transpose time part of the overhead function gives
$N^2 \log N \propto P^{4/3} \Rightarrow N \propto P^{2/3}$ will be sufficient.
$N^2 \log N \propto N^2 P^{1/3} \Rightarrow N \propto 2^{P^{1/3}}$.

The isoefficiency function is $f_{cube}(P) = O(P^{1/3} 2^{2P^{1/3}})$, and the memory required by every processor to maintain the efficiency must grow at the order of $O(\frac{2^{2P^{1/3}}}{P})$. Notice that $f_{cube}(P) < f_{mesh}(P) < f_{ring}(P)$. This implies that the Cube Exchange algorithm scales the best, the Mesh Exchange algorithm comes second and the Ring Exchange algorithm is the worst.

These isoefficiency functions are very large functions. To maintain parallel efficiency, N must grow exponentially in P. This, of course, is completely unrealistic for large P. Therefore, for all practical purposes the algorithms considered here are not scalable. In the following, we present a simpler analysis of the scalability of the algorithms considered, resulting in the same conclusion.

A close look at the computational complexity reveals that T_1 is almost linear with respect to the number of grid-points for realistic grid sizes. The higher order term dominates only if $N > 2^{80}$. Practically, $N \ll 2^{80}$, therefore $T_1 \sim O(N^2)$. Let t_1 be the compute time per grid-point per time-step for the shallow-water model. Assuming $N = M$, we can approximate the elapsed time for the sequential algorithm by

$$T_1 = t_1 N^2. \tag{4.22}$$

We can assume t_1 is independent of the problem size.

To maintain the parallel efficiency, the global communication costs (as one term of the overhead function) must be kept in proportion to the problem size W, i.e.

$$N^2 \propto N^2 P. \tag{4.23}$$

for the Ring Exchange algorithm. This is impossible, regardless how fast the problem size grows. Therefore, no isoefficiency function exist, and so the parallel system is not scalable.

Similarly, the proportionality between W and T_o requires

$$N^2 \propto N^2 P^{1/2} \tag{4.24}$$

for the Mesh Exchange algorithm, and

$$N^2 \propto N^2 P^{1/3} \tag{4.25}$$

for the Cube Exchange algorithm. Both relations are impossible regardless how fast the problem size grows. The global transposition time always grows faster than the computing time as P increases. Therefore an isoefficiency function does not exist, and so the parallel systems are not scalable.

As we will see later, it is realistic to assume that

$$t_s \sim t_1$$

$$t_w \sim 0.01 t_s$$

With this assumption, when $P \sim N$, we have $t_1 \cdot \frac{N^2}{P} \sim t_s \cdot P$. This implies that the startup overhead time will be of the same order as the computing time for every processor. Also for $100 < P < 1000$, we have $t_w \cdot P \sim t_s$. This implies the startup overhead is actually equivalent to the data transfer cost. Reducing the number of messages from $O(P)$ for the Ring Exchange algorithm to $O(P^{1/2})$ for the Mesh Exchange algorithm and $O(P^{1/3})$ for the Cube Exchange algorithm should improve the parallel efficiency.

Note that the phase 1 to phase 2 data redistribution requires $O(P^{1/2})$ messages, so the communication startup overhead is equivalent to or greater than the global transposition operation when the Mesh Exchange or the Cube Exchange algorithm is applied to the global transposition operation.

Nevertheless, the communication cost of the Mesh and Cube Exchange algorithms is of lower order than that of the Ring Exchange algorithm. We expect the Mesh and Cube Exchange algorithms to have better performance than the Ring Exchange algorithm. The Direct Exchange algorithm should have similar behavior to the Ring Exchange algorithm.

Chapter 5

Numerical Experiments and Results

The parallel shallow-water model is implemented using the Message Passing Interface (MPI). The implementation of the parallel model is based on the shallow-water code developed by Steve Thomas at Environment Canada [18]. The MPI Standard is a set of specifications for a message-passing library interface, developed by the MPI Forum, a group of researchers from government labs, universities and companies. MPI is becoming a widely accepted message-passing interface. The numerical experiments were carried out on Cyclone, a Cray T3E machine with 512 PEs, and an ethernet-connected workstation network at the Department of Computer Science, University of Toronto.

The Cray T3E is a MIMD parallel machine and provides shared physical address space of up to 2048 processors over a 3D torus interconnect. The architecture of the machine can be characterized as distributed shared-memory (DSM) where a global address space is supported through remote get/put operations. Distributed memory programming interfaces, such as MPI and PVM, are supported. Each node of the system contains an Alpha 21164 processor, system control chip, local memory and network router. The interconnection network forms a three dimensional torus that connects the nodes in the x,y and z dimensions. Figure 5.1 illustrates the periodic 3D torus interconnection network of the Cray T3E. MPI for the Cray T3E is an implementation of MPI derived from $MPI(EPCC)$, which was developed at the Edinburgh Parallel Computing Center in collaboration with Cray Research Inc.

The Computing Disciplines Facility (CDF) workstation cluster is a set of workstations connected by an ethernet. All workstations used in our numerical experiments have the same system configuration, which includes one 'UltraSparc Model 140' CPU and a system memory of 64Mb. $MPICH$ is used as the message passing environment on the CDF workstation cluster. $MPICH$ is a freely available, complete implementation of the MPI standard.

Performance results for the parallel shallow-water model using different transposition algorithms are presented in the following sections. The transposition algorithms are incorporated into the direct FFT solver for the Helmholtz equations. The elapsed times required by the transposition problem, the Helmholtz solver and shallow-water model are reported. This includes computing time, communication time, idle time and all other overhead. The results for the sequential model are also reported to verify that the parallel model reproduces the results of the sequential solver and to study the parallel performance.

Figure 5.2 is a plot of the shallow-water free surface on a 50 x 51 grid. Figure 5.2.a shows the surface height at time $t = 0$. Figure 5.2.b is the result at $t = 48000$ seconds.

5.1 Calibration of the Performance Model

To calibrate the parallel performance models, we need to find the machine dependent quantities

t_1: the elapsed time per grid-point per time-step for the shallow-water model;

t_s: the startup time of one message passing operation; and

t_w: the transfer time per floating-point number for the message passing operation.

The elapsed time per grid-point per time-step for the shallow-water model is obtained by running the sequential model on one processor several times. A different number of grid-points is used in each run. Figure 5.3 shows the elapsed times (for 160 time steps) versus the grid-point number for the sequential model. The figure confirms the linear approximation to the computing time:

$$T_1 = t_1 \cdot M \cdot N. \tag{5.1}$$

From the numerical results we get $t_1 \simeq 1.2122 \times 10^{-5}$. This is essentially independent of the problem size $M \cdot N$.

MPI provides point-to-point communication routines that implement the transfer of data between a pair of processes as determined by the matching of a send and a receive operation. The point-to-point communication routines are used to implement the Direct, Ring, Mesh and Cube Exchange algorithms. MPI also offers a rich set of collective communication routines which operate over a group of processors. We have tested an all-to-all collective communication routine, where each process sends distinct data to each of the receivers in the process group. This routine can be readily used to implement the transposition algorithms.

To measure the communication parameters t_s and t_w for MPI, a standard 'ping-pong' test was carried out. In this test, processor A sends a message to processor B. Upon receiving the message, processor B sends the message back to A. This process is repeated 1000 times and the aggregate time is divided by 2000 to get the time for one message passing operation. Figure 5.4 shows the elapsed time of the MPI message passing operation against the message size measured in floating-point numbers. The message size ranges from 0 to 1K floating-point number in increments of 10.

Figure 5.4 shows that there is a jump in the communication time at 512 floating-point numbers. This corresponds to the MPI internal message packet-size.

The numerical results verify the linear relationship between the communication time and the message size if the message size is between 0 and 512 floating-point numbers.

Consider the linear model discussed in Chapter 2,

$$t_{comm} = t_s + mt_w, \tag{5.2}$$

where t_s is the startup time, t_w is the per floating-point number transfer time and m is the message size. The parameters t_s and t_w in this model are computed to fit the observed data in a least-squares sense.

The startup time for sending and receiving a message is $t_s = 1.3178 \times 10^{-5}$ seconds, and $t_w = 7.1162 \times 10^{-8}$ seconds/floating-point number when the message size $m \leq 512$ floating-point numbers. For a message size larger than 512 floating-point numbers, $t_s = 3.0729 \times 10^{-5}$ and $t_w = 5.1730 \times 10^{-8}$. These results confirm the assumption made in Chapter 4 about the relationships between t_1, t_s and t_w on the Cray T3E.

Similar experiments were carried out on the CDF machines. By running the sequential shallow-water model on CDF, we found that the elapsed time per grid-point per time-step for the sequential shallow-water model is $t_1 \simeq 1.5312 \times 10^{-2}$ seconds. The 'ping-pong' test shows that the startup time per message is $t_s \simeq 3.3874 \times 10^{-4}$ seconds, and the transfer rate is $t_w \simeq 7.7755 \times 10^{-6}$ seconds. These parameters can be used to estimate the communication cost used in the parallel shallow-water model. However, care must be taken when interpreting these values. The 'ping-pong' test, which sends the same data (using the same data buffer) between two processes, allows data to reside entirely in the memory cache. In our parallel shallow-water model, neither the sending buffer nor the receiving buffer is mapped into the cache, and this situation can affect the performance of the communication. Another important factor is the number of processors used. Even if only two are communicating, the communication parameters t_s and t_w may increase as the number of processors that are used in the application increases. This is due to overhead associated with buffering and with deadlock detection. We expect faster communication on the two-processor ping-pong test than that on the parallel shallow-water model.

5.2 Numerical Results

In this section we present the performance results of the parallel shallow-water model using different transposistion algorithms on the Cray T3E and the CDF workstation cluster.

To investigate the scalability of the algorithms, numerical experiments are carried out for various problem sizes and number of processors.

Here '2D transposition' means the data redistribution from $phase2$ to $phase3$ and then back to $phase2$, which involves all-to-all data communication.

For the Direct Exchange algorithms, the data sent from one processor to every other processor are independent of each other. There is no requirement for synchronization between the messages. We used two communication modes to implement the Direct Exchange algorithm. Scheme A refers to the Direct Exchange algorithm using the asynchronous mode routines $MPI\_ISend()$ and $MPI\_IRecv()$. Each process posts all the requests for receipt of messages using $MPI\_IRecv()$, then sends out the data by calling $MPI\_ISend()$. These routines return immediately. The call to $MPI\_WaitAll()$ is used to make sure that all the processors have received the messages successfully. This is coded as follows,

```
do i = 1, P
  MPI_IRecv(...)
enddo
do i = 1, P
  MPI_ISend(...)
enddo
MPI_WaitAll(...)
```

where P is the number of processors used.

Scheme D refers to the Direct Exchange algorithm using the standard blocking mode point-to-point communication routines. Each process sends to all other processes using $MPI\_Send()$ and receives from all other processes using $MPI\_Recv()$. This is coded as follows,

```
do i = 1, P
  MPI_Send(...)
enddo
do i = 1, P
```

```
    MPI_Recv(...)
  enddo
```

The standard blocking mode routines $MPI\_Send()$ and $MPI\_Recv()$ are used in the Ring, Mesh and Cube Exchange algorithms as well. These algorithms are referred to as the 'R', 'M' and 'C' schemes, respectively, in the tables.

Also an MPI collective communication interface $MPI\_AllToAll()$ is used to implement the global transposition problem, referred to as scheme 'L' in the tables.

Timing the performance of the parallel algorithm is non-trivial. The difficulties include the following.
1) The machine may be time-shared by several other users. This is especially troublesome on the CDF workstation cluster. The communication channels may be shared by other users. The timing results are not very accurate, and may vary significantly from run to run.
2) Load imbalance and message buffering in the exchange of messages provided by MPI may cause the timing results to vary on different processors.

To minimize the inaccuracy, the numerical experiments were carried out when the system was lightly loaded. To measure the time required by each component of the shallow-water model, some explicit synchronization was used before and after the component. In each integration step, all the processes were synchronized before timing the starting point of the component that was to be measured. The maximum and average values of the individual elapsed time of this component are taken across all the processors. These values are accumulated over all time steps.

The process of measuring the elapsed time of a component A in the shallow-water model can be illustrated as follows,

```
....
MPI_Barrier();
tstart = MPI_WTime();
component A;  /* the component (computing or communication) to be measured */
tend   = MPI_WTime();
```

```
etime  = tend - tstart;
take the maximum and average of 'etime' across the PEs,
accumulate the maximum and average of 'etime'.
....
```

This is how the timing results of the 'Transpose', '1D-2D', and 'Computation' parts of the Helmholtz solver were collected. This technique is used to time the whole Helmholtz solver and shallow-water model as well. Of course, when measuring the Helmholtz solver or the shallow-water model, the unnecessary synchronization within it is removed. For a given problem size and number of processors, we need 3 runs to get these timing results. The first run gives the three components of the Helmholtz problem. The second and the third runs give the results for the Helmholtz solver and the shallow-water model, respectively.

The maximum value should be close to the elapsed time since there are synchronizations (implicitly) before and after every component. There is quite a big difference between the maximum and the average of the 'transpose' time, especially for the results for the Direct Exchange algorithm. The reason is that the communication channel is shared among all of the processes, and so some processes may be able to finish communicating much earlier than others. This is more likely for the Direct Exchange algorithm, since the data to be sent by one process is independent of the data sent by all others. While for the Ring, Mesh and Cube Exchange algorithms, every process must ensure that the required data have been received before sending the data to the next neighbor processor. This reduces the overall elapsed time differences among the processes.

Since all the processes must synchronize (implicitly) before proceeding to the next component, the maximum value should be the closest to the real elapsed time for the parallel application. In all the test cases, 160 integration steps are taken. The elapsed-times are accumulated over the 160 time steps.

5.2.1 Results on the Cray T3E

Tables 5.1 - 5.3 show the results on Cyclone for problem sizes 256×256, 512×512 and 1024×1024 grid-points. Up to 512 PEs are used on Cyclone. The first column gives the number of processors in the form of $P_x \times P_y \times P_z$, where P_x, P_y and P_z are the number of PEs in the x, y and z directions. The second column shows the corresponding algorithms used in the global transposition problem for the Helmholtz solver.

All three components of the Helmholtz solver are listed in the tables. The 'transpose' column gives the elapsed time for the global transpose operation portion of the Helmholtz solver. The '1D-2D' column gives the elapsed time for the data redistribution between Phase 1 and Phase 2. The 'Computation' column is the computing time, which includes the elapsed time for the FFTs and tridiagonal solves. 'Helmholtz Solver' includes all three parts. The last column is the elapsed time for the whole shallow-water model.

Figures 5.5 – 5.16 show graphically the parallel performance of the transposition algorithms, the Helmholtz solver and the whole parallel shallow water model. Except for Figure 5.16, the plots are in a log-log scale. Figure 5.5 shows the elapsed time for the global transpose algorithms versus the number of processors P for the problem size 256×256 grid-points. We can see that the elapsed time for the Direct Exchange algorithm and the Ring Exchange algorithm increases almost linearly as P increases when the number of PEs is relatively large. On the other hand, the elapsed time for the Mesh and Cube Exchange algorithms as well as for the $MPI\_AllToAll()$ routine are significantly lower than those for the Direct Exchange and the Ring Exchange algorithms.

When P is small, the $MPI\_AlltoAll$ routines gives the best performance. But when P is large, the Mesh and Cube Exchange algorithms are faster.

The elapsed time for the $MPI\_AllToAll$, Mesh and Cube Exchange algorithms decreases as P increases for small P, and then increases as P continues to increase. The elapsed time for the Mesh and Cube Exchange algorithms has a smaller increase rate with respect to P than the $MPI\_AllToAll$ routine, with the Cube Exchange algorithm having the smallest rate. This means that the Cube Exchange algorithm scales the best against the increase in PEs.

Figure 5.6 shows the performance of the Helmholtz solver, and Figure 5.7 shows the performance of the parallel shallow-water model. The performance differences of the Helmholtz solver are significant since the global transpose operation quickly becomes the dominate part as P increases. The same is also true for the whole shallow-water model as P increases. Figure 5.8 shows how the parallel efficiency changes as P increases. Obviously, the parallel efficiency decreases as P increases.

Figures 5.9 – 5.16 show similar results for the larger problem sizes of 512×512 and 1024×1024, respectively. At first glance, the efficiencies in Figure 5.12 and 5.16 show a strange behavior. The parallel efficiencies are greater than 1 for a small number of processors. In the case of problem size 1024×1024, the parallel efficiencies increase when the number of processors increases from 8 to 16 processors. The reason for this is the cache effect. The smaller the number of processors, the larger the data to be allocated to each processor, the larger the data to be exchanged during transpositions, and the worse the utilization of the cache. Depending on the size of the cache, there are cache misses when copying received or local data from the message buffer into the transposed field or when copying the local data into the transposed field. For a smaller number of processors, these cache effects are more dominant than the communication overhead. The cache effects are reduced with an increased number of processors - but then the influence of the communication becomes more and more dominant.

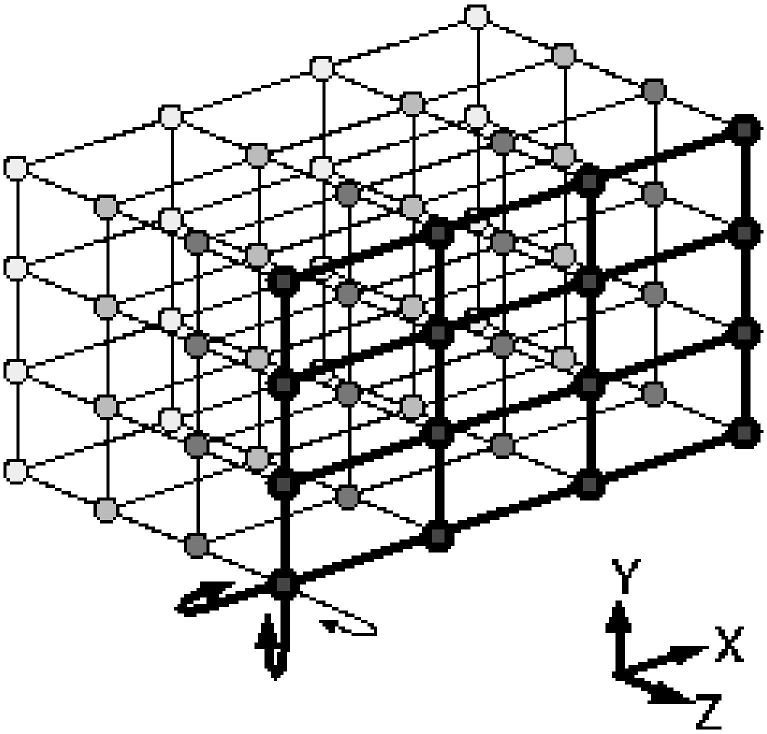

Figure 5.1: 3D torus interprocessor connection.

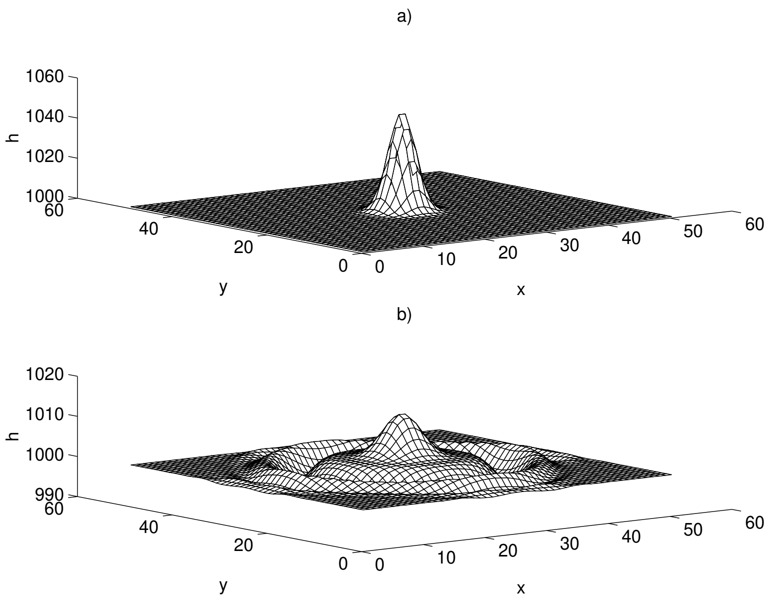

Figure 5.2: The 2D shallow-water free surface. a) Initial data, b) Data after integration at $t = 48000$ seconds.

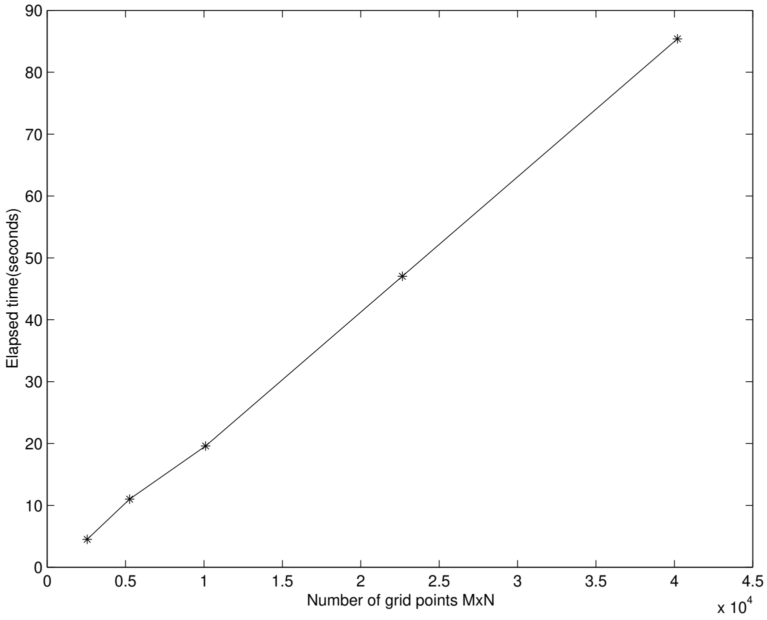

Figure 5.3: Elapsed time of the sequential 2D shallow-water model run on one processor of the Cray T3E.

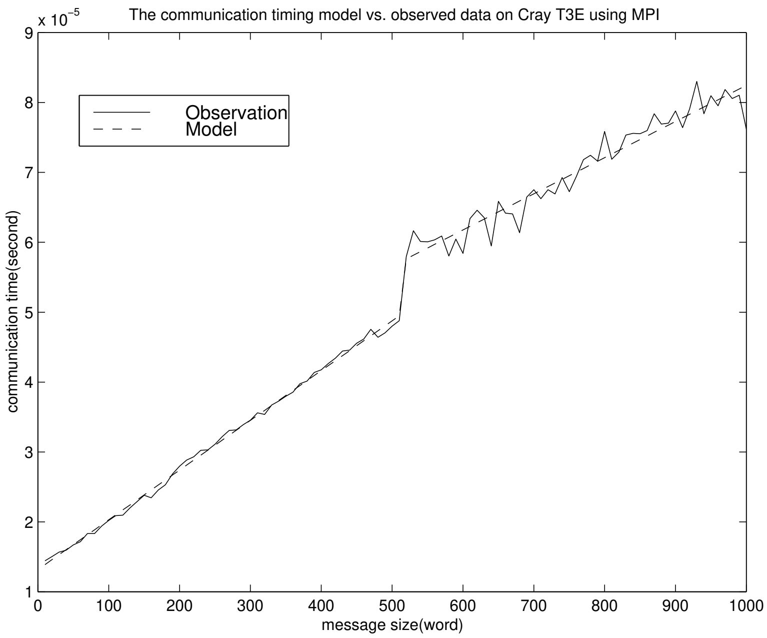

Figure 5.4: Communication time per message vs. the message size on the Cray T3E using MPI.

PEs	Algorithm	Transpose		1D - 2D		Computation		Helmholtz Solver		Shallow Water
		max	avg	max	avg	max	avg	max	avg	Model
2x2x2	A	1.6487	1.5707	1.2672	1.1449	3.6471	3.5042	6.3112	6.2259	19.7564
	D	1.7456	1.6246	1.1927	1.1169	3.6489	3.5038	6.2351	6.1428	19.6451
	R	1.4711	1.4421	1.1904	1.1160	3.6446	3.5033	6.1793	6.1197	19.5945
	M	2.1169	2.0685	1.1865	1.1167	3.6479	3.5054	6.7839	6.7287	20.1816
	C	2.5367	2.4765	1.1889	1.1173	3.6504	3.5055	7.1789	7.1203	20.5807
	L	0.9510	0.8652	1.1802	1.1106	3.6533	3.5151	5.6156	5.5560	19.0159
2x2x4	A	1.9939	1.7908	1.0068	0.8703	1.4844	1.3862	3.9458	3.8387	11.0149
	D	1.8937	1.7002	0.9606	0.8500	1.4789	1.3858	3.7642	3.6616	10.8077
	R	1.3692	1.3350	0.9590	0.8518	1.4815	1.3917	3.7637	3.7077	10.8037
	M	1.5512	1.4936	0.9584	0.8529	1.4756	1.3804	3.8152	3.7570	10.8220
	C	1.7261	1.6667	0.9592	0.8534	1.4745	1.3808	3.9827	3.9222	11.0080
	L	0.7431	0.6917	0.9584	0.8528	1.4728	1.3784	3.0215	2.9641	10.0546
2x4x4	A	2.6165	2.2647	0.8099	0.6327	0.7456	0.6422	3.4045	3.2534	7.0219
	D	2.3525	2.0417	0.7228	0.5973	0.7466	0.6432	3.0525	2.9209	6.6412
	R	1.7333	1.6647	0.7283	0.5981	0.7449	0.6392	3.0418	2.9734	6.6893
	M	1.3773	1.2709	0.7233	0.5989	0.7406	0.6413	2.6360	2.5717	6.2767
	C	1.3757	1.2761	0.7269	0.5988	0.7452	0.6415	2.6510	2.5841	6.3030
	L	0.6881	0.6217	0.7272	0.5979	0.7439	0.6378	1.9640	1.9004	5.5818
4x4x4	A	5.0991	4.2864	0.7767	0.5785	0.4197	0.3208	4.9482	4.6781	6.8255
	D	4.5112	3.8491	0.7704	0.5748	0.4180	0.3200	4.2583	4.0587	6.0825
	R	3.1908	3.0759	0.7591	0.5694	0.4113	0.3167	4.2556	4.1487	6.2796
	M	1.2755	1.1034	0.7769	0.5760	0.4198	0.3189	2.3253	2.1887	4.2823
	C	1.2856	1.1121	0.7870	0.5788	0.4250	0.3208	2.3098	2.1766	4.2907
	L	0.9505	0.8749	0.7745	0.5757	0.4138	0.3158	1.9458	1.8348	3.8858
4x4x8	A	10.7363	8.8990	0.7341	0.4785	0.2709	0.1738	9.1620	8.6053	9.8013
	D	9.3681	7.8569	0.7091	0.4725	0.2666	0.1731	7.5686	7.1644	8.2627
	R	6.3930	6.2433	0.6842	0.4649	0.2513	0.1680	7.1969	7.0722	8.2639
	M	1.5985	1.3756	0.7285	0.4730	0.2680	0.1697	2.4256	2.2548	3.4963
	C	1.4352	1.1991	0.7494	0.4771	0.2756	0.1725	2.1996	2.0461	3.3535
	L	1.8110	1.6712	0.7313	0.4764	0.2695	0.1703	2.5232	2.3868	3.6702
4x8x8	A	24.3932	21.1834	1.5859	0.8726	0.1955	0.1006	19.0191	17.7388	19.2568
	D	22.2637	19.8354	1.1715	0.7797	0.1964	0.1001	16.4676	15.9244	17.3758
	R	13.0617	12.8642	1.1002	0.7636	0.1741	0.0954	14.0390	13.8677	14.6806
	M	2.0821	1.7289	1.1701	0.7783	0.1906	0.0975	3.1849	2.9306	3.9269
	C	1.7219	1.4178	1.1793	0.7852	0.1952	0.0979	2.8499	2.6171	3.6031
	L	3.0608	2.8901	1.1892	0.7848	0.1978	0.0980	4.0580	3.8502	4.7318

Table 5.1: Timing results from the parallel shallow-water model on the Cray T3E for problem size 256×256.

PEs	Algorithm	Transpose		1D - 2D		Computation		Helmholtz Solver		Shallow Water
		max	avg	max	avg	max	avg	max	avg	Model
2x2x2	A	5.0960	4.7718	4.3353	4.0314	33.7851	33.6088	42.3280	42.0137	96.6433
	D	5.6801	5.2785	4.1888	3.9992	33.7887	33.6118	42.7402	42.3687	96.8745
	R	4.2464	4.1920	4.2164	4.0093	33.7771	33.6022	41.9168	41.7810	96.1310
	M	6.9024	6.8409	4.1934	3.9916	33.7959	33.6110	44.5335	44.4014	98.7251
	C	8.6096	8.5596	4.1875	3.9964	33.7969	33.6172	46.2151	46.0883	100.3974
	L	3.0915	2.9173	4.1979	3.9989	33.7788	33.6038	40.7214	40.5109	94.9434
2x2x4	A	3.5732	3.3503	2.9985	2.6109	16.1083	15.8826	21.8597	21.5804	48.3059
	D	4.0826	3.7222	2.9472	2.5865	16.1105	15.8886	21.9233	21.5979	48.3372
	R	3.2567	3.1999	2.9442	2.5831	16.0966	15.8742	22.1068	21.9219	48.5413
	M	4.6968	4.5812	2.9363	2.5763	16.1102	15.8797	23.3417	23.1547	49.7573
	C	5.5454	5.4075	2.9305	2.5739	16.1030	15.8803	24.2018	24.0193	50.6167
	L	1.9814	1.8749	2.9369	2.5779	16.1073	15.8850	20.8948	20.6813	47.3417
2x4x4	A	4.9646	4.3819	1.9381	1.6234	6.2318	5.9989	11.2050	10.9393	25.1994
	D	4.8979	4.3525	1.8762	1.5894	6.2370	6.0018	11.0936	10.8098	25.0669
	R	3.3256	3.2293	1.8726	1.5870	6.2414	6.0040	11.2441	11.1084	25.2454
	M	3.4172	3.3013	1.8700	1.5848	6.2407	5.9974	11.0984	10.9721	25.1194
	C	3.7329	3.5954	1.8697	1.5850	6.2370	5.9964	11.4507	11.3180	25.4610
	L	1.8169	1.7020	1.8718	1.5873	6.2420	5.9984	9.5496	9.4189	23.5873
4x4x4	A	6.7664	5.7392	1.7008	1.3124	1.7341	1.4821	7.4158	7.0948	14.5738
	D	6.2211	5.3760	1.6044	1.2686	1.7383	1.4857	6.9115	6.6004	13.9423
	R	4.3608	4.2152	1.6138	1.2701	1.7313	1.4851	7.2702	7.1032	14.4706
	M	2.7215	2.5489	1.6002	1.2662	1.7303	1.4788	5.5678	5.3996	12.7684
	C	2.8466	2.5774	1.6029	1.2685	1.7237	1.4760	5.6041	5.4404	12.7939
	L	1.5417	1.4364	1.6050	1.2684	1.7368	1.4798	4.3585	4.2013	11.5631
4x4x8	A	12.4842	10.4120	1.2264	0.9008	0.8624	0.6491	10.9968	10.3833	14.2480
	D	11.1510	9.4702	1.2248	0.8998	0.8562	0.6514	9.6910	9.1645	12.8970
	R	7.7102	7.5360	1.2039	0.8958	0.8393	0.6440	9.6009	9.4215	13.2123
	M	2.7173	7.5360	1.2250	0.9042	0.8477	0.6425	4.3987	4.2108	8.1195
	C	2.4553	2.2322	1.2224	0.9051	0.8516	0.6431	4.3987	4.2108	7.8142
	L	1.8298	1.7062	1.2230	0.8999	0.8486	0.6456	3.5058	3.3300	7.2269
4x8x8	A	27.2796	23.4801	1.7897	1.1453	0.5148	0.3281	21.5824	20.6310	22.4107
	D	24.8368	22.1133	1.5242	1.0586	0.5139	0.3314	18.8084	18.2076	19.7459
	R	14.4747	14.2528	1.5030	1.0506	0.4913	0.3259	16.0173	15.8135	17.7782
	M	3.1130	2.6705	1.5516	1.0620	0.5076	0.3269	4.7509	4.4502	6.5978
	C	2.7193	2.3613	1.5343	1.0648	0.5114	0.3256	4.3203	4.0466	6.2860
	L	3.5709	3.3589	1.5458	1.0631	0.5199	0.3284	5.1516	4.9065	7.0153
8x8x8	A	78.8110	71.0197	2.1414	1.1580	0.3662	0.1888	55.3639	53.3837	51.7260
	D	74.7051	68.9088	1.4844	0.9469	0.3633	0.1897	50.2765	49.0212	49.1796
	R	29.4731	29.1472	1.4462	0.9417	0.3546	0.1858	30.2383	29.9678	31.3783
	M	4.2439	3.7209	1.4823	0.9512	0.3630	0.1848	5.7336	5.3532	6.6796
	C	3.2142	2.7263	1.5034	0.9557	0.3653	0.1836	4.5171	4.1886	5.6435
	L	6.0145	5.7403	1.5432	0.9615	0.3804	0.1882	7.3153	7.0048	8.3454

Table 5.2: Timing results from the parallel shallow-water model on the Cray T3E for problem size 512×512.

PEs	Algorithm	Transpose		1D - 2D		Computation		Helmholtz Solver		Shallow Water
		max	avg	max	avg	max	avg	max	avg	Model
2x2x2	A	18.7073	17.3634	15.9655	15.1599	235.4498	235.0128	267.2159	266.0544	488.9422
	D	20.1154	18.4154	15.7286	15.0655	235.4712	235.0030	268.3968	266.9523	489.3444
	R	14.3055	14.1992	15.8236	15.1019	235.4594	235.0046	265.2724	264.8379	486.0808
	M	25.1394	25.0409	15.7861	15.0872	235.4657	235.0051	275.8466	275.4499	496.9553
	C	31.9597	31.8541	15.7814	15.0969	235.4628	235.0082	282.4832	282.0913	503.4104
	L	10.9661	10.5255	15.7711	15.0957	235.4552	234.9887	262.4341	261.5845	483.5650
2x2x4	A	11.6200	10.7022	10.8181	9.3571	91.8601	91.4261	110.5845	109.4089	205.7349
	D	13.2880	12.0346	10.7573	9.2803	91.8461	91.4252	111.3510	110.1386	206.0253
	R	9.2153	9.1249	10.7537	9.2920	91.8491	91.4248	111.2836	110.4991	206.1528
	M	16.0052	15.7089	10.7319	9.2721	91.8572	91.4309	117.2367	116.5066	212.0728
	C	19.3815	18.9783	10.7997	9.3149	91.8638	91.4281	120.6479	119.9129	215.2683
	L	6.9998	6.6606	10.7595	9.2777	91.8332	91.4032	109.0447	108.2015	203.8249
2x4x4	A	8.1208	7.4317	6.4394	5.3469	48.9140	48.4543	65.6017	64.8403	105.1640
	D	9.3975	8.4998	6.4190	5.3150	48.9275	48.4644	65.9600	65.0847	105.3846
	R	7.0283	6.9199	6.4214	5.3287	48.9019	48.4494	66.9004	66.4903	106.2390
	M	10.0883	9.9217	6.4137	5.3104	48.9164	48.4573	69.3260	68.8861	108.7012
	C	11.5752	11.3755	6.4259	5.3239	48.9170	48.4534	71.0304	70.6109	110.4284
	L	5.2086	4.8714	6.4291	5.3371	48.9047	48.4454	64.6267	64.1900	104.0493
4x4x4	A	12.6044	10.9596	4.5287	3.5164	24.1308	23.6907	34.5715	33.8472	55.1193
	D	12.4227	10.9480	4.4616	3.4760	24.1352	23.6960	34.2932	33.5572	55.7626
	R	7.2876	7.1115	4.4698	3.4760	24.1439	23.6980	35.3688	34.8643	56.0122
	M	6.6953	6.5034	4.4519	3.4695	24.1356	23.6905	34.4656	33.9419	55.0657
	C	7.9110	7.6830	4.4631	3.4748	24.1433	23.6922	35.4406	34.9030	55.0657
	L	4.2728	4.0610	4.4569	3.4730	24.1372	23.6896	31.9518	31.4635	52.5498
4x4x8	A	16.5158	13.7880	3.1435	2.2394	7.1804	6.6952	19.2372	18.4869	32.8115
	D	15.2854	13.0323	3.0118	2.1666	7.1795	6.6998	18.3543	17.5455	31.8003
	R	9.8466	9.6075	3.0017	2.1616	7.1779	6.6961	19.2912	18.8862	33.4187
	M	5.1146	4.8371	3.0002	2.1612	7.1737	6.6912	14.5272	14.1063	28.6057
	C	5.4096	5.1626	3.0031	2.1620	7.1739	6.6913	14.6926	14.2655	28.7659
	L	3.1975	2.9853	3.0054	2.1673	7.1797	6.6905	12.4718	12.0274	26.6275
4x8x8	A	33.1598	28.7196	2.9481	2.1781	2.0399	1.6206	26.2157	25.1587	32.1983
	D	30.6997	27.6437	2.7351	2.1062	2.0520	1.6259	24.4227	23.4466	30.6489
	R	17.3204	17.0252	2.7130	2.1044	2.0292	1.6208	21.4061	21.0789	28.4346
	M	4.8932	4.4199	2.7338	2.1118	2.0318	1.6191	8.8064	8.4466	15.8923
	C	4.3712	4.0069	2.7280	2.1110	2.0478	1.6166	8.3031	7.9776	15.4493
	L	3.6809	3.4851	2.7244	2.1059	2.0346	1.6214	7.7311	7.3726	14.8309
8x8x8	A	80.1056	72.1618	2.8398	1.7288	1.0727	0.7324	56.6965	54.4802	55.8987
	D	77.2683	71.0389	2.2248	1.5328	1.0860	0.7384	53.1702	51.6934	54.5099
	R	32.1656	31.7846	2.2068	1.5279	1.0668	0.7317	35.7015	35.3094	39.0510
	M	5.8854	5.2117	2.2123	1.5330	1.0773	0.7311	8.4847	8.0301	12.1179
	C	4.3751	3.7653	2.2650	1.5387	1.0856	0.7296	6.6611	6.2629	10.2760
	L	7.7537	7.3483	2.2287	1.5335	1.0792	0.7341	10.3004	9.9125	13.7559

Table 5.3: Timing results from the parallel shallow-water model on the Cray T3E for problem size 1024×1024.

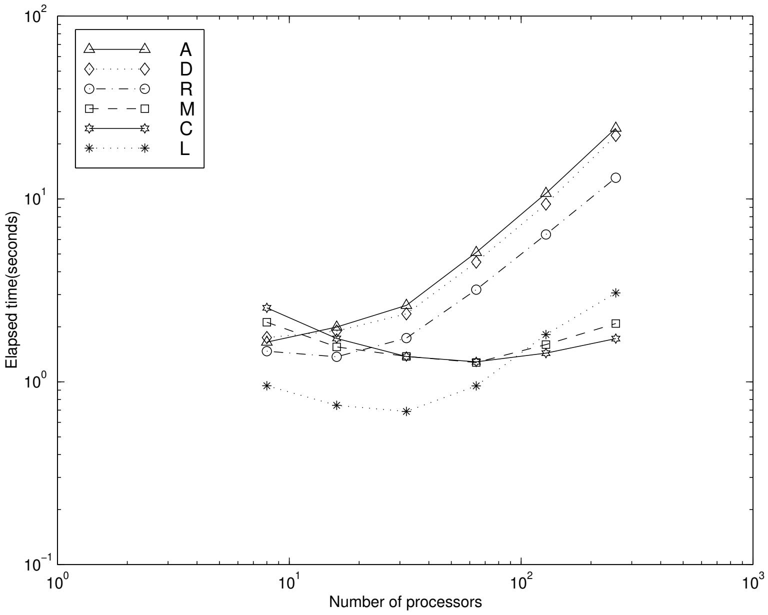

Figure 5.5: Performance of the transposition algorithms on the Cray T3E with 256×256 grid points.

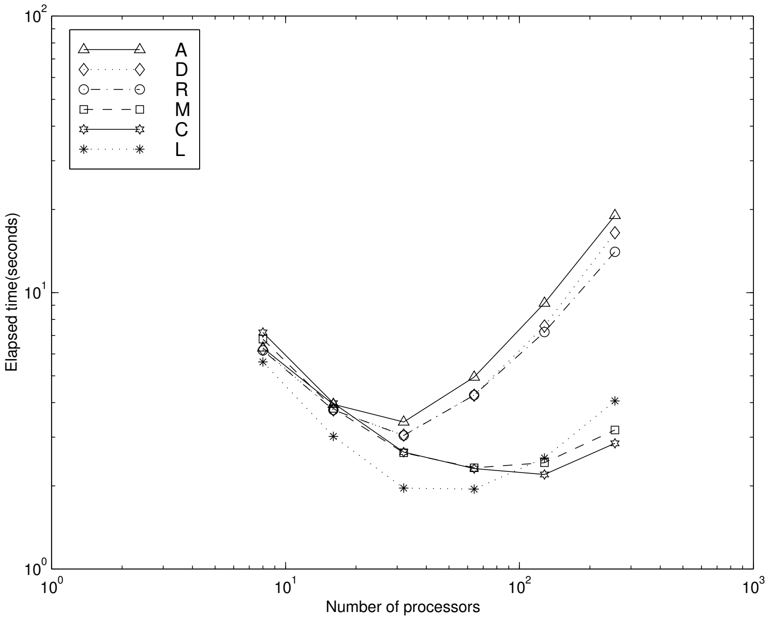

Figure 5.6: Performance of the Helmholtz solver using different transposition algorithms on the Cray T3E with 256×256 grid points.

Because of the communication intensive global transposition required by the direct Helmholtz solver, the parallel efficiency of this method decreases quickly. Nevertheless, the more sophiscated algorithms – the Mesh and Cube Exchange algorithms and

PEs	Algorithm	Transpose		1D - 2D		Computation		Helmholtz Solver		Shallow Water
		max	avg	max	avg	max	avg	max	avg	Model
8	A	26.0202	17.3982	25.9850	15.9120	9.8868	8.8096	46.0346	42.6256	82.7437
	D	22.0773	16.0587	20.4418	14.5576	9.2148	8.5436	42.6266	39.0361	79.5758
	R	25.0382	22.8208	21.9159	14.8111	10.4829	8.7079	47.8899	44.5895	86.0452
	R	28.9638	25.1416	22.2444	14.7340	9.7644	8.6198	52.1162	48.4499	88.8065
	C	32.4532	27.4199	20.6013	14.5210	9.5504	8.5950	54.6700	51.4256	91.8980
	L	23.5169	15.8593	20.5775	14.5481	9.3937	8.5719	41.9130	38.4459	78.9217
16	A	22.4880	14.0216	18.8317	11.4306	5.0936	4.3493	43.9981	37.5581	64.3946
	D	22.4579	13.7188	18.4446	11.2257	4.9910	4.3362	43.8389	36.1754	63.8311
	R	24.3618	22.5542	19.0429	11.3890	5.1032	4.3434	49.4513	45.3900	67.9433
	M	23.3729	19.6290	19.0550	11.2855	5.1008	4.3414	41.8852	38.0918	61.2207
	C	25.4104	21.6435	18.6810	11.2525	5.1149	4.3501	43.6869	39.4769	63.7877
	L	20.4114	12.7970	18.3821	11.1803	5.0992	4.3446	39.3261	33.0625	58.2084
27	A	56.6640	18.1422	18.3850	9.0251	3.3127	2.7181	71.2072	51.3668	78.1761
	D	37.9783	17.8285	17.3885	8.3570	3.3107	2.6204	59.0587	45.3334	64.9320
	R	53.8585	48.9064	17.8483	8.5023	3.3026	2.6191	73.8062	70.4043	79.4751
	M	30.3621	22.3183	17.6119	8.4359	3.6055	2.7019	39.0206	34.1418	55.6333
	C	24.9440	19.0940	17.1105	8.3905	3.4620	2.6286	38.4539	33.0012	51.7370
	L	36.8476	16.7967	16.8191	8.5746	3.5100	2.6301	53.8633	42.7499	70.7338
36	A	276.3002	54.1735	19.1410	7.7820	3.4525	2.1171	285.8038	257.6223	369.6019
	D	273.1330	72.5957	18.4618	7.9395	3.5806	2.1116	275.5515	196.6324	345.1695
	R	79.6331	72.7588	18.6434	7.6665	3.2912	2.0962	98.3798	87.7996	169.3615
	M	29.4509	22.9140	18.2591	7.7624	3.2509	2.0934	48.5640	45.9247	120.4023
	C	27.8006	20.1669	18.4900	7.5000	3.2752	2.0942	47.6733	43.0525	113.2460
	L	259.1341	58.1398	17.4241	7.9231	3.3583	2.1042	277.0028	248.1603	330.4155

Table 5.4: Timing results from the parallel shallow-water model on the CDF workstation cluster for problem size 150×150.

the *MPI_AllToAll* algorithm – have significantly better performance than the simple ones – the Direct and Ring Exchange algorithms. Moreover, when the number of PEs is large, the Cube Exchange algorithm becomes the best global transposition method.

5.2.2 Results on the CDF Workstation Cluster

Table 5.4 shows the performance results for a 150×150 grid-point shallow-water model on the CDF workstation cluster. The maximum number of processors used is 36. Figures 5.17 - 5.20 show graphically the parallel performance of the transposition algorithms, the Helmholtz solver and the whole shallow-water model. The global transposition becomes the dominant task in the whole shallow-water model when the number of processors is large. It is clear that the more sophiscated algorithms – the Mesh and Cube Exchange algorithms – scale better than the Direct Exchange and the *MPICH* provided *MPI_AlltoAll*() routine.

The parallel shallow-water model does not scale well beyond 8 processors with a problem size of 150×150 grid points.

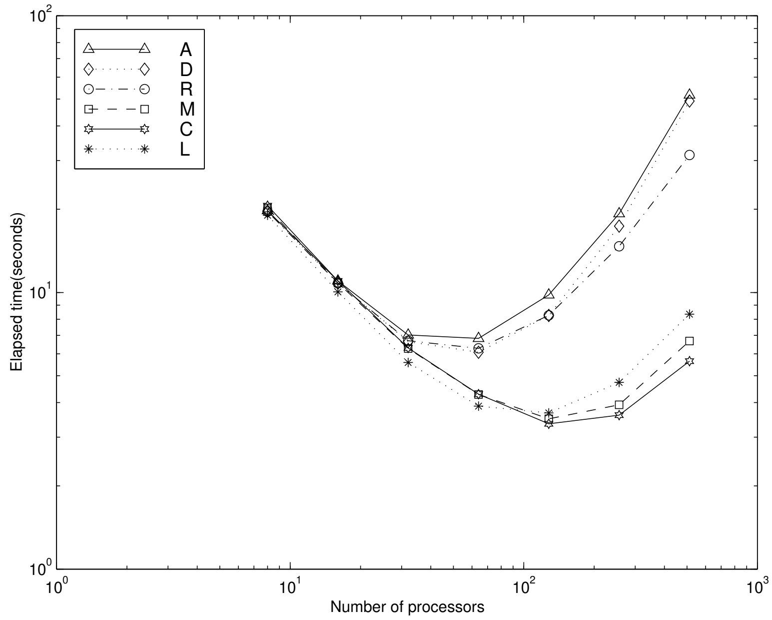

Figure 5.7: Performance of the shallow-water model using different transposition algorithms on the Cray T3E with 256×256 grid points.

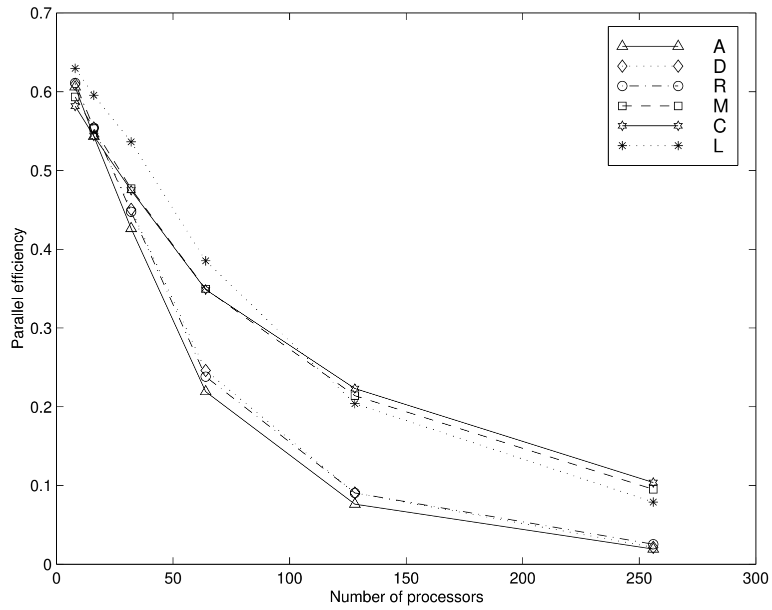

Figure 5.8: Parallel efficiency of the shallow-water model as a function of the number of PEs on the Cray T3E with 256×256 grid points.

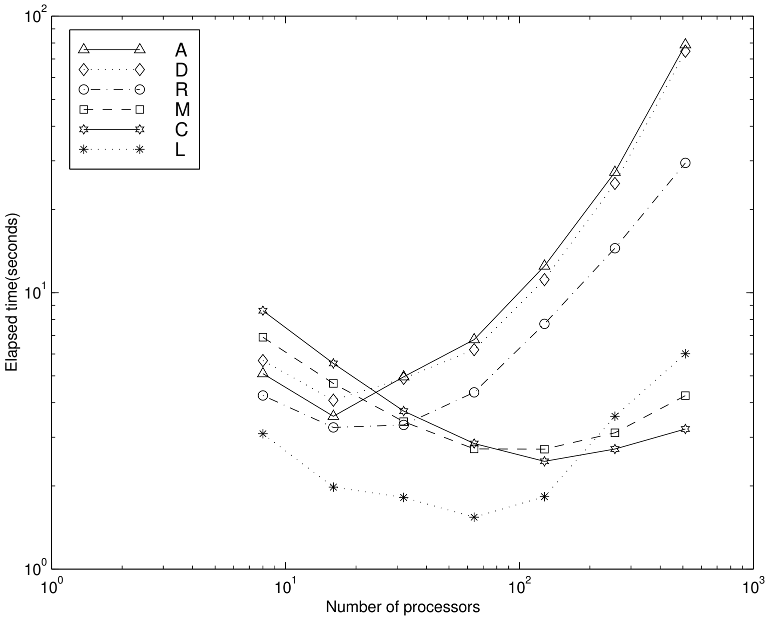

Figure 5.9: Performance of the transposition algorithms on the Cray T3E with 512×512 grid points.

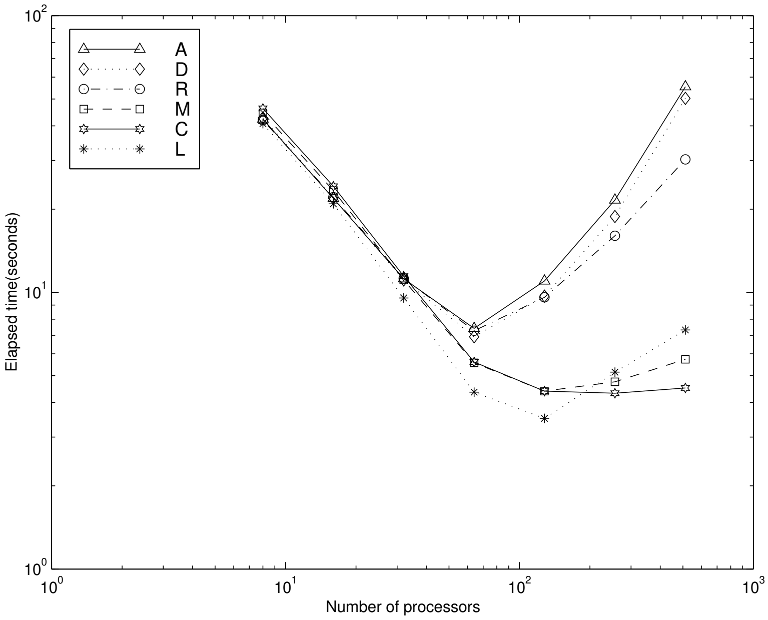

Figure 5.10: Performance of the Helmholtz solver using different transposition algorithms on the Cray T3E with 512×512 grid points.

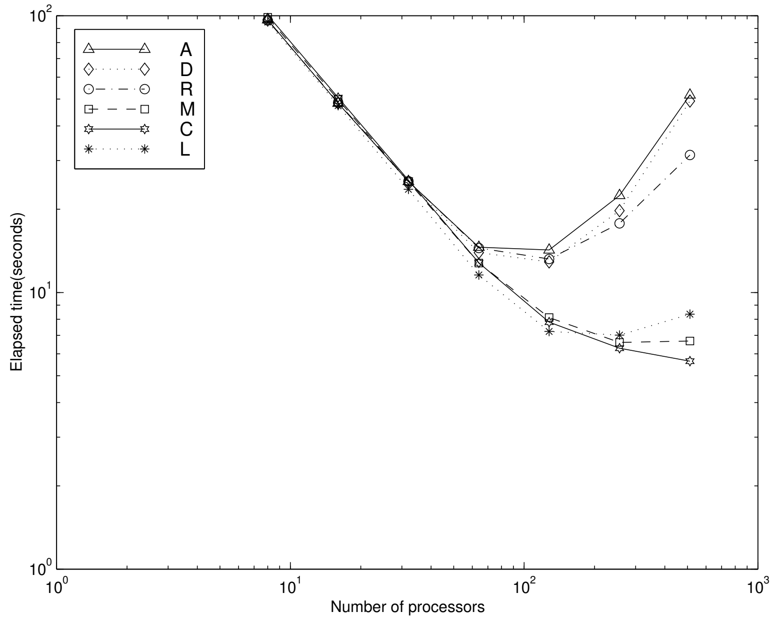

Figure 5.11: Performance of the shallow-water model using different transposition algorithms on the Cray T3E with 512×512 grid points.

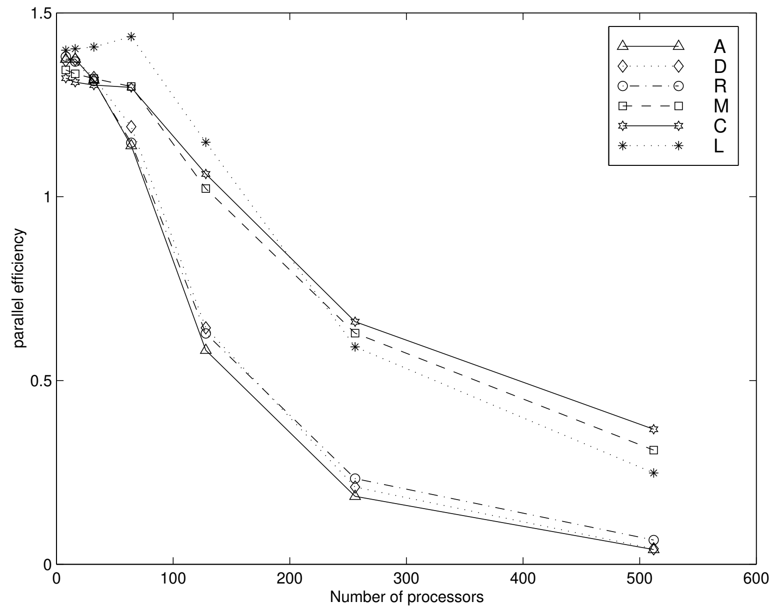

Figure 5.12: Parallel efficiency of the shallow-water model as a function of the number of PEs on the Cray T3E with 512×512 grid points.

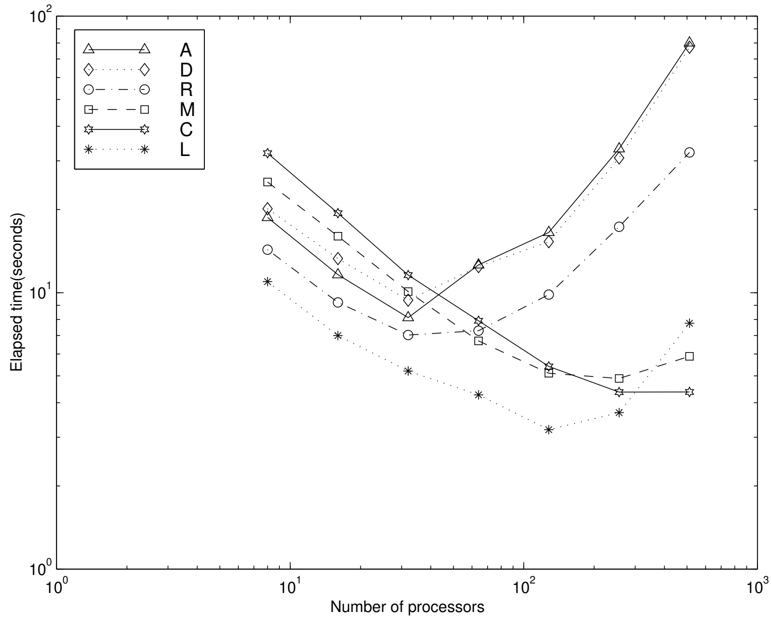

Figure 5.13: Performance of the transposition algorithms on the Cray T3E with 1024×1024 grid points.

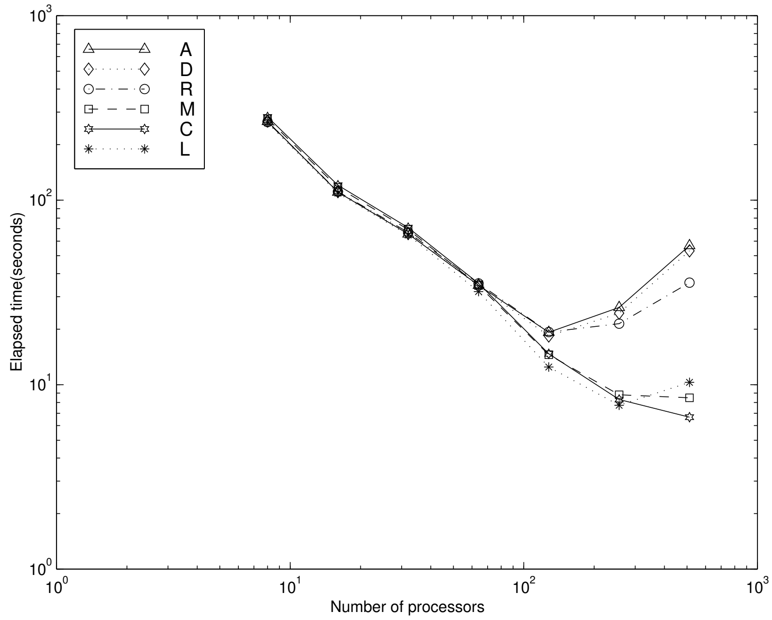

Figure 5.14: Performance of the Helmholtz solver using different transposition algorithms on the Cray T3E with 1024×1024 grid points.

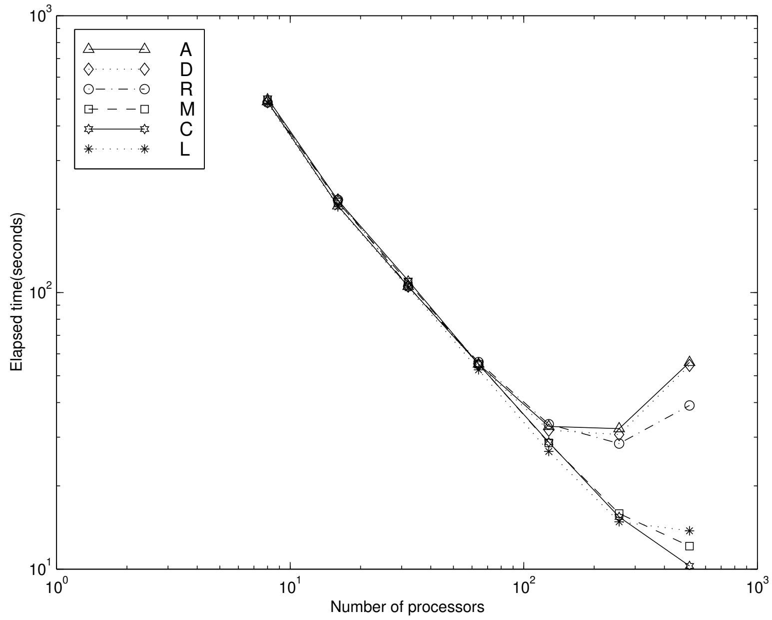

Figure 5.15: Performance of the shallow-water model using different transposition algorithms on the Cray T3E with 1024×1024 grid points.

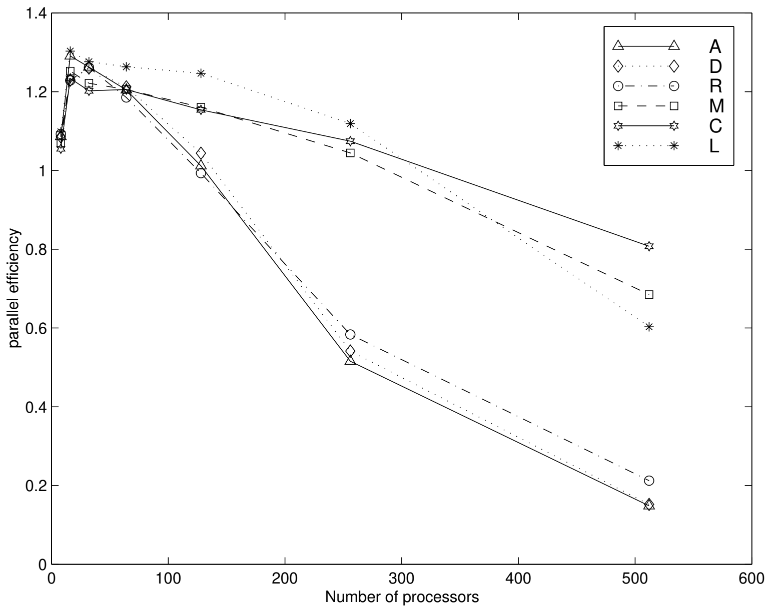

Figure 5.16: Parallel efficiency of the shallow-water model as a function of the number of PEs on the Cray T3E with 1024×1024 grid points.

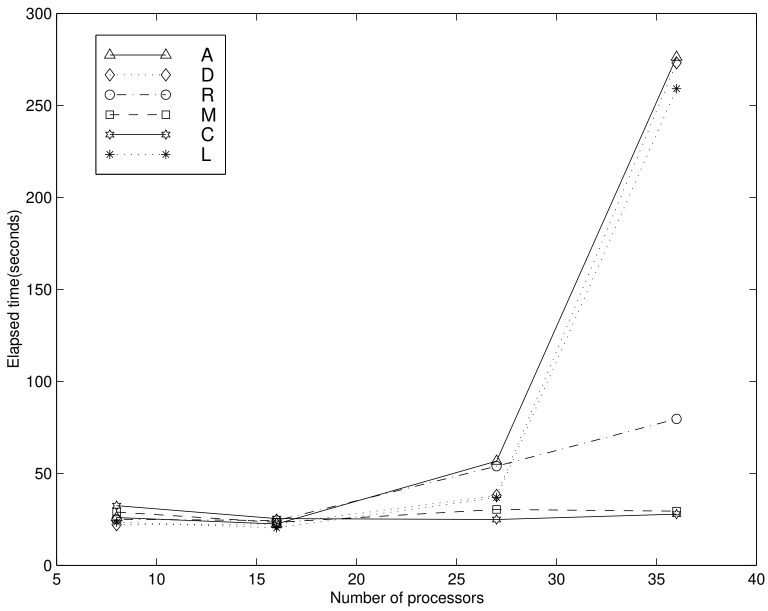

Figure 5.17: Performance of the transpose algorithms on the CDF workstation cluster with 150×150 grid points.

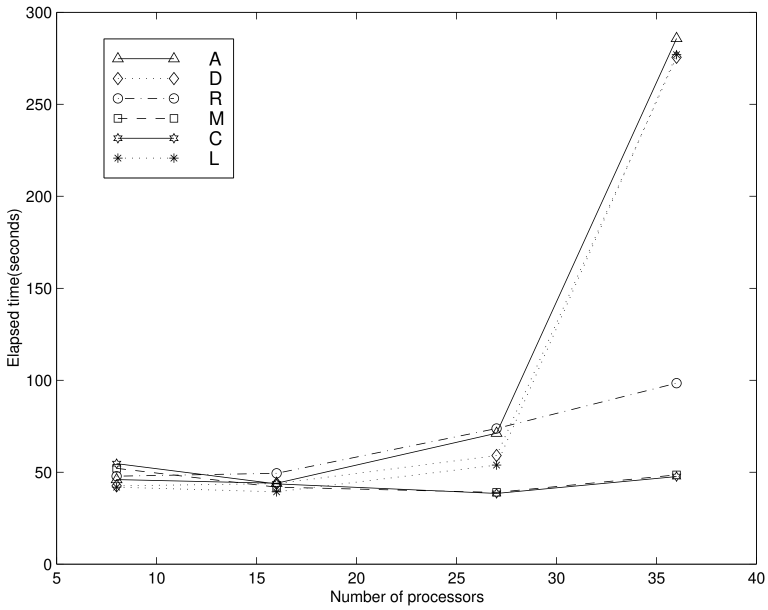

Figure 5.18: Performance of the Helmholtz solver using different transpose algorithms on the CDF workstation cluster with 150×150 grid points.

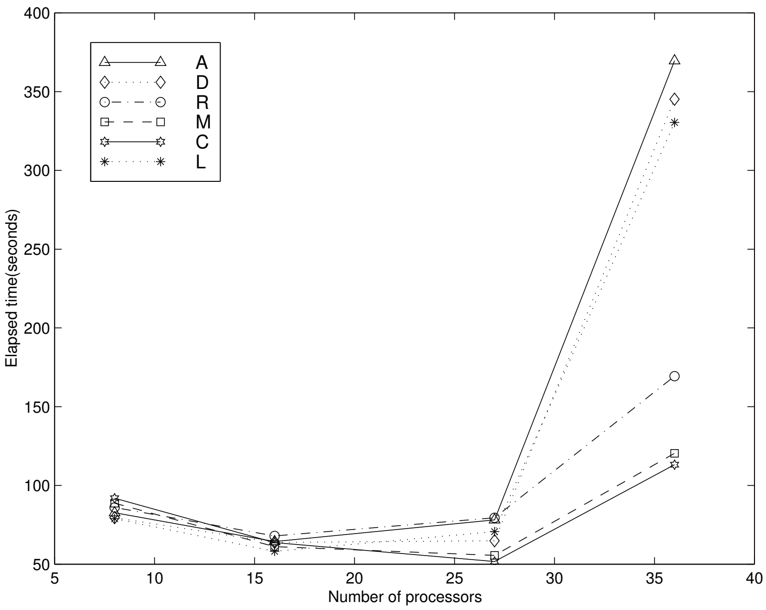

Figure 5.19: Performance of the shallow-water model using different transpose algorithms on the CDF workstation cluster with 150×150 grid points.

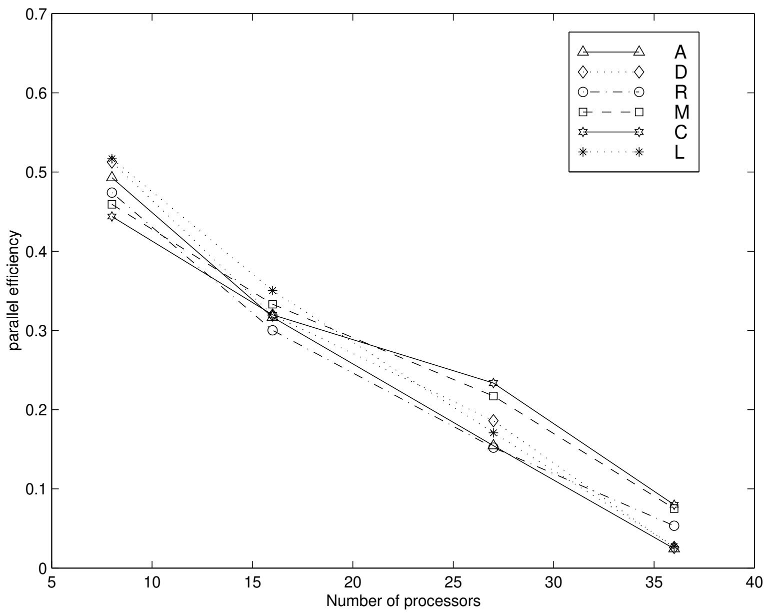

Figure 5.20: Parallel efficiency of the shallow-water model as a function of the number of processors on the CDF workstation cluster with 150×150 grid points.

Chapter 6

Conclusions

We have examined a parallel implementation of the shallow-water model using a direct Helmholtz solver. The global transposition task arising in the direct Helmholtz solver is found to be the bottleneck for the scalability. A few alternative algorithms are proposed for the global transposition task in addition to the commonly used methods. Based on a simple linear timing model of the message passing operation and the assumption that the communication startup cost associated with each message is relatively expensive, we have developed the more sophiscated algorithms – the Mesh Exchange and Cube Exchange algorithms. The goal is to reduce the number of messages by taking advantage of the topology of the communication network. The numerical results confirm that this helps to reduce the communication time.

The numerical results also show that this approach to solving the shallow water model is efficient on a parallel machine, such as the Cray T3E, when the number of processors used is not too large. However, due to the communication intensive global transposition task, the parallel efficiency drops quickly when the number of processors increases. Theoretically, this approach to solving the shallow-water model is not scalable, because the communication costs increase faster than the computing cost as the number of processors increases. Numerical experiments show that optimizing the communication portion in this parallel approach is necessary. Indeed, by using the more sophiscated algorithms, the communication cost for the global transposition task can be reduced significantly. It is important to keep in mind that the topology of the communication network has a significant impact on the scalability of the parallel system. As we have noticed, the ratio of the startup cost of the point-to-point communication and the computing cost on the CDF workstation is smaller than it is on the Cray T3E. However, numerical results show that the Cray T3E is more scalable. One reason for this that the Cray T3E has more communication links, which provide more communication capacity and scalability.

Further research can be done to improve the parallel shallow-water model. The Mesh and Cube Exchange algorithms were implemented using high level MPI communication routines. The performances may be improved if more efficient lower level services are used.

Appendix

Definitions and Some Properties of Tensor Products of Matrices

Let $A \in \mathbb{R}^{M \times N}$ and $B \in \mathbb{R}^{K \times L}$. The *tensor product* $A \otimes B$ (Kronecker product) of A and B is defined by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1N}B \\ a_{21}B & a_{22}B & \dots & a_{2N}B \\ \dots & \dots & \dots & \dots \\ a_{M1}B & a_{M2}B & \dots & a_{MN}B \end{bmatrix}$$

i.e. $(A \otimes B)_{ij} = a_{mn} \cdot b_{kl}$, when $i = (m-1)K + k$ and $j = (n-1)L + l$.

Some properties of the tensor product operator are

1. $(A + B) \otimes C = A \otimes C + B \otimes C$;
2. $A \otimes (B + C) = A \otimes C + A \otimes C$;
3. $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$;
4. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$.

Bibliography

[1] S.R.M. Barros. On the Parallelization of Global Spectral Eulerian Shallow-Water Models. *Parallel Supercomputing*, (World Scientific, 1993), G.R. Hoffman and T. Kauranne. eds., pp. 36-43.

[2] J. Choi, J. Dongarra and D. Walker. Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers. *Parallel Computing*, vol. 21, 1995, pp. 1387-1405.

[3] Cray Research Inc. MPP Software Guide (October 1994), SG-2508, 1.2.

[4] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

[5] Ananth Y. Grama, Ashul Gupta and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology*, vol. 1, no. 3, August 1993, pp. 12-21.

[6] R.W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *J. ACM*, vol. 12, 1965, pp. 95-133.

[7] S.L. Johnsson and C. Ho. Algorithms for Matrix Transposition on Boolean N-Cube Configured Ensemble Architectures. *SIAM J. Matrix Analysis and Application*, vol. 9(3), July 1988, pp. 419-454.

[8] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[9] David M. Nicol and Shahid H. Bokhari. Optimal Multiphase Complete Exchange on Circuit-Switched Hypercube Architectures. *SIGMETRICS, ACM*, vol. 22, no. 1, May 1994, pp. 252-260.

[10] A. Robert. A Stable Numerical Integration Scheme for the Primitive Meteorological Equations. *Atmos. Ocean*, vol. 19, 1981, pp. 35-46.

[11] A. Robert. A Semi-Lagragian and Semi-implicit Numerical Integration Scheme for the Primitive Meteorological Equations. *Jpn. Meteor. Soc.*, vol. 60, 1982, pp. 319-325.

[12] Roar Skålin. Scalability of Parallel Grid Point Limited Area Atmospheric Models I & II. *Manuscript, Dept. of Math. Sciences, Norwegian University of Science and Technology*, Trodheim, Norway, 1996.

[13] Roar Skålin. Scalability of Parallel Grid Point Limited Area Atmospheric Models. *J. Atmospheric and Oceanic Technology*, vol. 14, June 1997, pp. 426-455.

[14] A. Staniforth and J. Côté. Semi-Lagrangian Integration Schemes for Atmospheric Models – A Review. *Monthly Weather Review*, vol. 119, no. 9, 1991, pp. 2206-2223.

[15] C. Temperton. Direct Method for the Solution of the Discrete Poisson Equation. *J. Comput. Phys.*, vol. 31, 1979, pp. 1-20.

[16] S. Thomas. Notes on the GEF Family of Shallow-Water Models. *RPN Manuscript*, July 1994.

[17] S. Thomas and J. Côté. Massively Parallel Semi-Lagrange Advection. *Simulation Practice and Theory*, vol. 3, 1995, pp. 223-238.

[18] S. Thomas, J. Côté, A. Staniforth, I. Lie and R. Skålin. A Semi-Implicit Semi-Lagrange Shallow-Water Model for Massively Parallel Processors. *Preceedings of the 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology*, Nov. 1994, pp. 407-423.

[19] D.L. Williamson, J.B. Drake, J.J. Hack, R. Jakob and P.N. Swarztrauber. A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry. *J. Comput. Physics.*, vol. 102, no. 1, 1992, pp. 211-224.