1 AN EFFICIENT TRANSPOSITION ALGORITHM FOR DISTRIBUTED MEMORY COMPUTERS

Christina Christara, Xiaoliang Ding and Ken Jackson

Computer Science Dept., University of Toronto, Toronto, Ontario M5S 3G4, Canada.*

{ccc, ding, krj}@cs.toronto.edu

Abstract:

Data transposition is required in many numerical applications. When implemented on a distributed-memory computer, data transposition requires all-to-all communication, a time consuming operation. 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. These data transposition schemes were incorporated into a parallel solver for the shallow-water equations. We compare the performance of these schemes with that of the Direct Exchange Algorithm and the MPI all-to-all communication routine, `MPI_AllToAll`. The numerical experiments were performed on a Cray T3E computer with 512 processors and on an ethernet-connected cluster of 36 Sun workstations. Both the analysis and the numerical results indicate that the more sophisticated Mesh and Cube Exchange algorithms perform better than either the simpler well-known Direct and Ring Exchange schemes or the `MPI_AllToAll` routine. We also generalize the Mesh and Cube Exchange algorithms to a d-dimensional mesh algorithm, which can be viewed as a generalization of the standard hypercube data transposition algorithm.

Keywords: data transposition, shallow-water equations, Cray T3E, cluster of workstations, MPI, FFT, Helmholtz equation, d-dimensional mesh of processors.

*This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Information Technology Research Centre (ITRC) of Ontario, and Communications and Information Technology Ontario (CITO). We also thank Cray Research Inc. for allowing us to use one of their T3E computers to obtain our numerical results.

1.1 INTRODUCTION

Data transposition is required in many numerical applications, including both Fast Direct Solvers and the Alternating Direction (ADI) Method for linear algebraic systems as well as Spectral Methods for partial differential equations (PDEs). As a specific example, we investigate in this paper the data transposition required for the parallel solution of the shallow-water equations by a semi-implicit semi-Lagrangian scheme on a distributed memory computer. A non-linear two-dimensional (2D) Helmholtz equation must be solved at every time-step of this scheme. Applying a fixed-point iteration to the non-linear Helmholtz equation requires the repeated solution of a linear 2D Helmholtz equation. An elegant and effective means of solving the linear algebraic system arising from the spatial discretization of the linear Helmholtz equation is to use a series of independent Fast Fourier Transforms (FFTs) in one direction followed by a series of independent tridiagonal system solves in the other direction. This process is commonly referred to as a Fast Direct Solver (VanLoan92; Thomas94).

On a distributed-memory parallel computer, it is highly desirable to have each processor perform one or more FFTs using data stored locally on that processor and similarly to perform one or more tridiagonal system solves using data stored locally on that processor. However, to perform a FFT, a processor must have all the data from a row of the discretized 2D domain, while, to perform a tridiagonal system solve, a processor must have all the data from a column of the discretized 2D domain. That is, data transposition is required between the FFT and tridiagonal system solve phases of the Fast Direct Solver. This implies that each processor must exchange data with each other, a potentially time-consuming task. Several researchers have considered this data transposition problem; see, for example, (Choi95; Johnsson88; Nicol94) and the references therein.

In this paper, we propose the Mesh Exchange and Cube Exchange data transposition algorithms designed to exploit, respectively, a two- or three-dimensional processor interconnection network. The algorithms proposed do not require any restriction on the size of the matrix to be transposed, nor on the processor layout. Moreover, they can be extended to a d-dimensional processor interconnection network (mesh), for any d. Furthermore, the algorithm for a d-dimensional mesh is applicable on any machine on which a d-dimensional mesh can be embedded.

A parallel numerical method for the shallow-water equations incorporating these data transposition schemes was implemented using MPI (Gropp94). The method was run on both a Cray T3E computer with 512 processors and an ethernet-connected cluster of 36 Sun workstations. The numerical results indicate that the more sophisticated Mesh and Cube Exchange algorithms perform better than either the simpler well-known Direct and Ring Exchange schemes or the MPI all-to-all communication routine `MPI_AllToAll`.

1.2 DESCRIPTION OF THE PROBLEM

In this section, we briefly review the shallow-water equations and an effective numerical scheme for solving them, with emphasis on the data exchange requirements of the numerical scheme. For a more detailed description of the equations and the numerical scheme, see (Ding98; Robert82; Thomas94).

The shallow-water equations are three time-dependent PDEs in two space dimensions that model the inviscid flow of a thin layer of fluid. More specifically, they are used to predict the horizontal and vertical velocities, u and v, respectively, of a fluid particle and its geopotential height ϕ at a point in a 2D domain. In this paper, we assume the domain is rectangular and the boundary conditions are periodic in both dimensions.

A semi-implicit two-time-level scheme for the shallow-water equations treats the advection terms explicitly, while the other terms are time-averaged between the next (arrival) and the current (departure) points in time. Such a scheme allows for relatively large time-steps, while maintaining stability. If a semi-implicit scheme is coupled with a semi-Lagrangian technique, the time-steps can be increased even further (Robert82). A semi-Lagrangian treatment of advection applies an implicit correction to the explicit scheme by tracing the trajectory backwards and computing the departure point using a fixed-point iteration.

To solve for u, v and ϕ at each time-step, we first eliminate u and v from the system to obtain a (scalar) non-linear 2D Helmholtz equation for ϕ. Applying a fixed-point iteration to solve the non-linear 2D Helmholtz equation requires the repeated solution of a linear 2D Helmholtz equation with an appropriate right side. In practice, only two iterations of this fixed-point iteration are required to solve the non-linear 2D Helmholtz equation for ϕ. We complete the time-step by computing u and v from ϕ.

Thus, the computation required by the semi-implicit semi-Lagrangian method for the solution of the shallow-water equations can be described as follows.

for each time-step
1. compute the right side of the semi-implicit discretized shallow-water equations at the current time-step
2. apply a fixed-point iteration to compute the departure points at the current time-step (semi-Lagrangian technique)
3. compute the right side of the semi-implicit discretized shallow-water equations at the departure points
4. compute the right side of the discretized linear 2D Helmholtz equation
5. solve the discretized linear 2D Helmholtz equation for a first approximation to ϕ
6. update the right side of the discretized linear 2D Helmholtz equation
7. solve the discretized linear 2D Helmholtz equation for ϕ
8. compute u and v from ϕ

end

We now concentrate on the description of the computation required for Steps 5 and 7 of the algorithm above. Each of these steps involves the solution of a linear algebraic system associated with the discretized linear 2D Helmholtz equation.

We assume that the rectangular domain of the problem is discretized into a regular $N \times M$ grid, where N is the number of grid-points in the x-direction and M is the number of grid-points in the y-direction, and that a linear spline Galerkin method is used for the spatial discretization. Then the matrix associated with the linear algebraic systems takes a tensor-product form (Ding98; Thomas94) and each linear algebraic system can be solved by applying a Fast Direct Solver. If the right side vector of the linear system is rearranged as a matrix with M rows and N columns, the Fast Direct Solver proceeds in three steps:

(i) apply a FFT to each row of the right-side matrix;
(ii) apply an almost tridiagonal system solve to each column of the matrix produced in (i);
(iii) apply an inverse FFT (iFFT) to each row of the matrix produced in (ii).

Within each step of the process described above, the FFTs, almost tridiagonal system solves and iFFTs can be performed concurrently. Therefore, on a distributed-memory parallel computer, it is highly desirable to have each processor perform one or more FFTs, almost tridiagonal system solves and iFFTs using data stored locally on that processor. However, to perform a FFT or iFFT, a processor needs to access all the data in a row of the discretized 2D domain, while, to perform an almost tridiagonal system solve, it needs to access all the data in a column of the discretized 2D domain. This raises the need to transpose the $M \times N$ data matrix that stores the values corresponding to grid-points.

Moreover, when solving the shallow-water equations on a distributed-memory parallel computer, it is advantageous to divide the domain of the problem into $d_x \times d_y$ subdomains (i.e., to use a 2D domain decomposition). Each processor is assigned the data and computation associated with the points of one subdomain. Such a decomposition is favorable during all steps of the shallow-water algorithm, except Steps 5 and 7. For further details, see (Ding98; Thomas94).

Thus, the parallel algorithm for solving the shallow-water equations has three different phases of data access modes:

Phase 1 requires a 2D x–y block partition of data;
Phase 2 requires a 1D x-direction block partition of data;
Phase 3 requires a 1D y-direction block partition of data.

Therefore, the parallel algorithm for solving the discretized linear 2D Helmholtz equation in Steps 5 and 7 of the shallow-water-equations algorithm proceeds as follows.

(a) Redistribute the data from the 2D x–y block partition (Phase 1) to a 1D x-direction block partition (Phase 2);
(b) Perform the M FFTs of size N in the x-direction;

(c) Transpose the data from the 1D x-direction block partition (Phase 2) to a 1D y-direction block partition (Phase 3);
(d) Solve the N almost tridiagonal systems of size M in the y-direction;
(e) Transpose the data from the 1D y-direction block partition (Phase 3) to a 1D x-direction block partition (Phase 2);
(f) Perform the M iFFTs of size N in the x-direction;
(g) Redistribute the data from the 1D x-direction block partition (Phase 2) to a 2D x–y block partition (Phase 1).

The redistribution of data between Phases 1 and 2 in Steps (a) and (g) does not require a global exchange of data among all processors. Rather, each processor must exchange data with the subset of processors assigned to the subdomains in the same row or column of the $d_x \times d_y$ domain decomposition. On the other hand, the redistribution of data between Phases 2 and 3 in Steps (c) and (e) requires global communication among all processors. That is, each processor must exchange data with all other processors. This is one of the most demanding communications tasks in terms of bandwidth and latency on distributed memory systems. A similar data transposition problem arises in several other numerical applications, including the Alternating Direction (ADI) Method for linear algebraic systems and Spectral Methods for PDEs.

Figure 1.1 illustrates the global communication required to transpose a data matrix on a 4-processor system. Arrows between blocks identify exchange-pairs involving processor P_0. Note that P_0 must send one block of data to each other processor and receive one block of data from each other processor. This data transposition task is costly; it is therefore important to implement it efficiently. In the following section, we describe various parallel algorithms for this task.

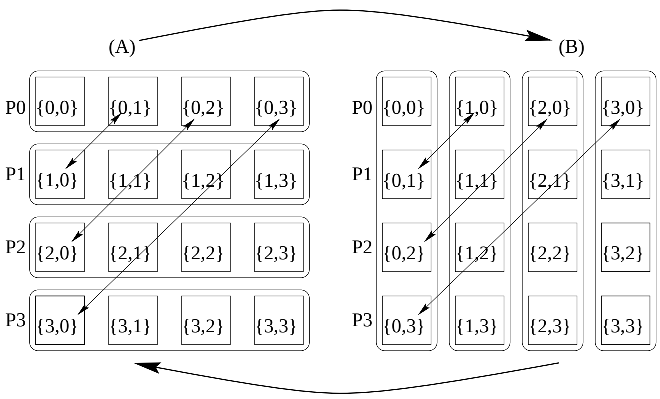

Figure 1.1 The data exchange pattern for processor P_0 to transpose a data matrix on a 4-processor system.

1.3 ALGORITHMS FOR THE DATA TRANSPOSITION TASK

In this section we present several algorithms for the (block) data transposition task and analyze their communication complexity. For the analysis, we adopt the following simple model for the communication latency when a message of size m is sent from a processor to its neighbour:

$$t_{\mathrm{comm}} = t_s + m t_w \tag{1.1}$$

where t_s is the startup cost and t_w is the per unit of datum transfer cost. (The unit is usually a byte, a word or a floating-point number.) For most parallel computers, $t_s \gg t_w$. Therefore, to keep the communication cost low, it is important to send as few messages as possible by combining small messages into larger ones whenever feasible.

Assume that P processors are available. For $p \in \{0, \ldots, P-1\}$, we denote processor p by P_p. Also assume that the data are subdivided into $P \times P$ blocks, where $\mathrm{blk}(i,j)$, $i = 0, \ldots, P-1$, $j = 0, \ldots, P-1$, has about M/P rows and N/P columns, and so each block is approximately of size $m_b = MN/P^2$. Note that P does not need to divide M or N exactly. In Phase 2, processor p holds blocks $\mathrm{blk}(p,j)$, $j = 0, \ldots, P-1$. In Phase 3, it holds blocks $\mathrm{blk}(i,p)$, $i = 0, \ldots, P-1$.

We start by describing the well-known *Direct Exchange* and *Ring Exchange* algorithms (Choi95; Nicol94). Each requires $\mathcal{O}(P)$ messages per processor. To reduce the number of messages sent by each processor, we extend the Ring Exchange algorithm to the *Mesh Exchange* and *Cube Exchange* algorithms that exploit, respectively, a two- or three-dimensional processor interconnection network. The Direct, Ring, Mesh and Cube Exchange algorithms are described below. For all algorithms, we show the pseudo-code run by the processor indexed 'me'.

1.3.1 The Direct Exchange Algorithm

The Direct Exchange algorithm is a straightforward way of performing data transposition. A processor exchanges a message with each of the other $P-1$ processors directly as depicted in Figure 1.2. The message size of each transfer is the size of each block (i.e., approximately m_b). Note that no synchronization is needed between two consecutive sends or receives. Therefore, the algorithm can be implemented using either asynchronous or synchronous sends and receives.

```
The Direct Exchange Algorithm
    for i = 1 to P-1 do
        precv = (me+i) mod P
        send blk(me, precv) to processor precv
    for i = 1 to P-1 do
        psend = (me-i) mod P
        receive blk(psend, me) from processor psend
```

If there is a direct link from each processor to each other, then the total communication cost predicted by the message-passing cost model (1.1) for each processor is

$$T_{\mathrm{dir}} \approx (P-1)(t_s + m_b t_w) \approx P t_s + \frac{MN}{P} t_w$$

since $m_b = MN/P^2$. In practice, however, a parallel machine may not have sufficiently many links to guarantee that each processor can exchange messages

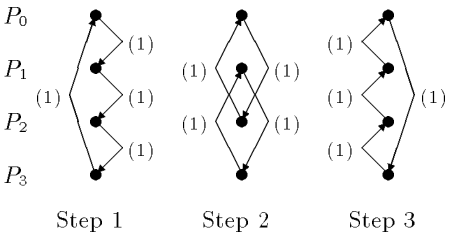

Figure 1.2 The Direct Exchange Algorithm on a 4-processor system. The number of blocks of data transmitted at each step is given in parenthesis.

with each other concurrently without contention. For a cluster of workstations connected by an ethernet, for example, all P processors share one communication channel. According to Foster (Foster95), the communication cost for such a cluster (and many other distributed memory computers) is approximated more accurately by

$$T_{\mathrm{dir}} \approx (P-1)(t_s + m_b P t_w) \approx P t_s + M N t_w \tag{1.2}$$

1.3.2 The Ring Exchange Algorithm

A linear array of processors with a wraparound connection is referred to as a *ring*. Since each processor in a ring communicates with its two neighbours only, a store-and-forward technique is used to route messages to their destination. To reduce the total number of messages, the data blocks to be sent are packed into one message whenever possible.

The Ring Exchange algorithm proceeds in $P-1$ steps as depicted in Figure 1.3. On step $i \in \{1, \ldots, P-1\}$, each processor sends a message containing $P-i$ blocks (where each block is of size approximately m_b) to the next processor in the ring and receives a message of about the same size from the previous processor. Then each processor extracts the one block it needs from the received message and prepares to forward the remaining $P-i-1$ blocks on the next step. Synchronization is required between steps to ensure that the processor has received its message before it tries to forward a part of it to the next processor.

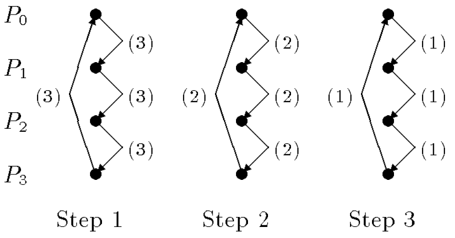

Figure 1.3 The Ring Exchange Algorithm on a 4-processor system. The number of blocks of data transmitted at each step is given in parenthesis.

```
The Ring Exchange Algorithm
   pnext = (me+1) mod P
   pprev = (me-1) mod P
   msg_send(1) = {blk(me, (me+i) mod P), i = 1,...,P-1}
   for i = 1 to P-1 do
     send msg_send(i) to pnext
     receive msg_recv(i) from pprev
     extract data block blk((me-i) mod P, me) from msg_recv(i)
     set msg_send(i+1) to the remaining blocks
```

The communication cost of this algorithm for each processor is

$$T_{\mathrm{ring}} \approx \sum_{i=1}^{P-1} (t_s + (P-i) m_b t_w) = (P-1)\left(t_s + \frac{P}{2} m_b t_w\right) \approx P t_s + \frac{MN}{2} t_w \quad (1.3)$$

where we have used $m_b = MN/P^2$. In total, each processor sends and receives $P-1$ messages. This is optimal for a ring with P processors and uni-directional links, since it takes $P-1$ hops to send a message around the ring. Moreover, this message count is asymptotically optimal for a ring with bi-directional links.

1.3.3 The Mesh Exchange Algorithm

A two-dimensional array of processors is called a *mesh*. Assuming the mesh has a wraparound connection along each row and column, each processor of the mesh communicates with its four neighbours only. Let P_x and P_y be the number of processors in the x and y directions respectively and note that $P = P_x P_y$. No other restriction is imposed on the choice of P_x or P_y.

When discussing the Mesh Exchange algorithm, it is convenient to index each processor by two subscripts, rather than one. More specifically, if $p = jP_x + i$ where $i \in \{0, \ldots, P_x - 1\}$ and $j \in \{0, \ldots, P_y - 1\}$, we use the alias $P_{i,j}$ for processor P_p. The processors in row i (i.e., processors $\{P_{i,j} : j = 0, \ldots, P_y - 1\}$) form a ring in the x-direction. Similarly, the processors in column j (i.e., processors $\{P_{i,j} : i = 0, \ldots, P_x - 1\}$) form a ring in the y-direction. Figure 1.4 shows the 16 processors of a 4×4 mesh and their connections.

The Mesh Exchange algorithm proceeds in two phases. In the first phase, the Ring Exchange algorithm is performed concurrently along each row of the mesh. This phase requires $P_x - 1$ steps. On step $i \in \{1, \ldots, P_x - 1\}$, each processor sends a message consisting of $P_x - i$ blocks, where each block is of size approximately $P_y m_b$, to the next processor in its row and receives a message of about the same size from the previous processor in its row. Then each processor extracts one block of size approximately $P_y m_b$ from the received message and prepares to forward the remaining $P_x - i - 1$ blocks on the next step. No communication is required between processors along the y-direction. At the end of the first phase, the processors $\{P_{i,j} : i = 0, \ldots, P_x - 1\}$ in column j of the mesh collectively have all the data blocks needed for the processors in column j, but the data blocks must be redistributed among the processors in column j.

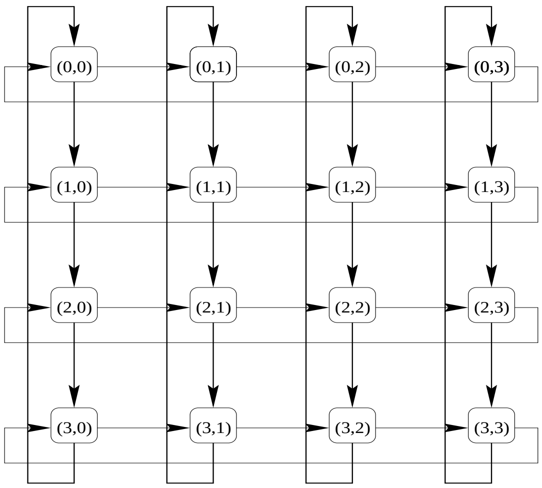

Figure 1.4 A 16-processor system arranged as a 4×4 mesh with wraparound connections. During the Mesh Exchange algorithm, the processors perform the Ring Exchange algorithm first along the rings in the x-direction simultaneously, then along the rings in the y-direction, again simultaneously.

In the second phase, the roles of rows and columns in the mesh are interchanged. The Ring Exchange algorithm is performed concurrently along each column of the mesh. This phase requires $P_y - 1$ steps. On step $j \in \{1, \ldots, P_y - 1\}$, each processor sends a message consisting of $P_y - j$ blocks, where each block is of size approximately $P_x m_b$, to the next processor in its column and receives a message of about the same size from the previous processor in its column. Then each processor extracts one block of size approximately $P_x m_b$ from the received message and prepares to forward the remaining $P_y - j - 1$ blocks on the next step. No communication is required between processors along the x-direction. At the end of the second phase, each processor has its required data.

```
The Mesh Exchange Algorithm
    perform the Ring Exchange algorithm in the x-direction
    perform the Ring Exchange algorithm in the y-direction
```

It follows from the communication model (1.3) for the Ring Exchange algorithm that the communication cost for the first phase of the Mesh Exchange algorithm is

$$T_x \approx (P_x - 1)\left(t_s + \frac{P_x}{2} P_y m_b t_w\right) = (P_x - 1)\left(t_s + \frac{MN}{2P} t_w\right)$$

where we have used $m_b = MN/P^2$ and $P = P_x P_y$. Similarly, the communication cost of the second phase is

$$T_y \approx (P_y - 1)\left(t_s + \frac{P_y}{2} P_x m_b t_w\right) = (P_y - 1)\left(t_s + \frac{MN}{2P} t_w\right).$$

Therefore, the total communication cost is

$$T_{\text{mesh}} = T_x + T_y \approx (P_x + P_y - 2)\left(t_s + \frac{MN}{2P} t_w\right). \tag{1.4}$$

In total, each processor sends and receives $P_x + P_y - 2$ messages. This is optimal for a mesh with $P_x P_y = P$ processors and uni-directional links, since it takes $P_x + P_y - 2$ hops to send a message from one corner of the mesh to the opposite corner. Moreover, this message count is asymptotically optimal for a mesh with bi-directional links.

If $P_x \approx P_y \approx \sqrt{P}$, then (1.4) reduces to

$$T_{\text{mesh}} \approx 2(\sqrt{P} - 1)\left(t_s + \frac{MN}{2P}t_w\right) \approx 2\sqrt{P}\,t_s + \frac{MN}{\sqrt{P}}\,t_w. \qquad (1.5)$$

1.3.4 The Cube Exchange Algorithm

A three-dimensional array of processors is referred to as a *cube*. A cube with wraparound connections along each dimension is referred to as a *torus*. Each processor of a torus communicates with its six neighbours only. Figure 1.5 shows the 64 processors of a $4 \times 4 \times 4$ torus with their connections.

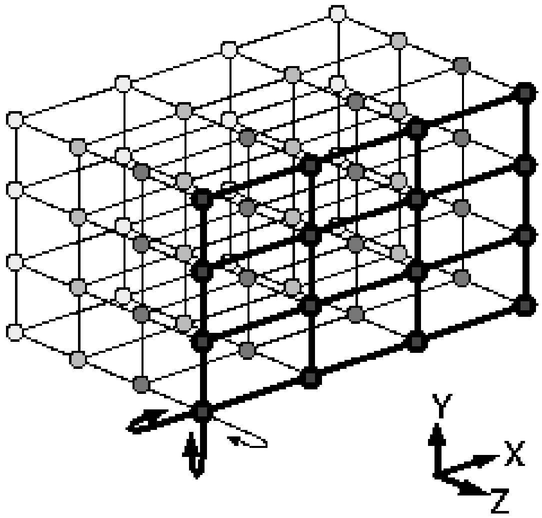

Figure 1.5 A 64-processor system arranged as a $4 \times 4 \times 4$ torus. (To avoid cluttering the figure, we have not included all wraparound connections.) During the Cube Exchange algorithm, the processors perform the Ring Exchange algorithm first along the rings in the x-direction simultaneously, then along the rings in the y-direction simultaneously, and finally along the rings in the z-direction, again simultaneously.

Let P_x, P_y and P_z be the number of processors in the x-, y- and z-directions, respectively. Note that $P = P_x P_y P_z$. No other restriction is imposed on the choice of P_x, P_y or P_z. When discussing the Cube Exchange algorithm, it is convenient to index the processors by three subscripts, rather than one. More specifically, if $p = kP_xP_y + jP_x + i$, where $i \in \{0, \ldots, P_x - 1\}$, $j \in \{0, \ldots, P_y - 1\}$ and $k \in \{0, \ldots, P_z - 1\}$, we use the alias $P_{i,j,k}$ for P_p. For fixed j and k, processors $\{P_{i,j,k} : i = 0, \ldots, P_x - 1\}$ form a ring in the x-direction. For fixed i and k, processors $\{P_{i,j,k} : j = 0, \ldots, P_y - 1\}$ form a ring in the y-direction. For fixed i and j, processors $\{P_{i,j,k} : k = 0, \ldots, P_z - 1\}$ form a ring in the z-direction. The Cube Exchange algorithm is based on applying the Ring

Exchange algorithm in each of the three dimensions, treating the processors along these directions as rings.

The algorithm proceeds in three phases. In the first phase, each x-direction ring performs the Ring Exchange algorithm concurrently. The first phase requires $P_x - 1$ steps. On step $i \in \{1, \ldots, P_x - 1\}$, $P_x - i$ data blocks, each of size approximately $P_y P_z m_b$, are sent from each processor to the next processor in the x-direction and each processor receives a message of about the same size from the previous processor in the x-direction. Then each processor extracts one block of size approximately $P_y P_z m_b$ from the received message and prepares to forward the remaining $P_x - i - 1$ blocks on the next step. No communication is required between processors along the y- or z-directions. At the end of the first phase, the processors $\{P_{i,j,k} : i = 0, \ldots, P_x - 1\}$ in each plane j, k of the cube collectively have all the data blocks needed for the processors in plane j, k, but the data blocks must be redistributed among the processors in plane j, k.

In the second and third phases of the Cube Exchange algorithm, the Ring Exchange algorithm is applied in a similar way to the rings in the y- and z-directions, respectively. For the sake of brevity, we leave the straightforward details to the reader.

```
The Cube Exchange Algorithm
    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
```

It follows from the communication model (1.3) for the Ring Exchange algorithm that the communication cost for the first phase of the Cube Exchange algorithm is

$$T_x \approx (P_x - 1)\left(t_s + \frac{P_x}{2} P_y P_z m_b t_w\right) = (P_x - 1)\left(t_s + \frac{MN}{2P} t_w\right)$$

where we have used $m_b = MN/P^2$ and $P = P_x P_y P_z$. Similarly, the communication costs of the second and third phases are

$$T_y \approx (P_y - 1)\left(t_s + \frac{MN}{2P} t_w\right) \qquad \text{and} \qquad T_z \approx (P_z - 1)\left(t_s + \frac{MN}{2P} t_w\right)$$

respectively. Therefore, the total communication cost is

$$T_{\text{cube}} = T_x + T_y + T_z \approx (P_x + P_y + P_z - 3)\left(t_s + \frac{MN}{2P} t_w\right). \tag{1.6}$$

In total, each processor sends and receives $P_x + P_y + P_z - 3$ messages. This is optimal for a cube with $P_x P_y P_z = P$ processors and uni-directional links, since it takes $P_x + P_y + P_z - 3$ hops to send a message from one corner of the cube to the opposite corner. Moreover, this message count is asymptotically optimal for a cube with bi-directional links.

If $P_x \approx P_y \approx P_z \approx \sqrt[3]{P}$, then (1.6) reduces to

$$T_{\text{cube}} \approx 3(\sqrt[3]{P} - 1)\left(t_s + \frac{MN}{2P} t_w\right) \approx 3\sqrt[3]{P}\, t_s + \frac{3}{2}\,\frac{MN}{P^{2/3}}\, t_w. \tag{1.7}$$

1.3.5 The d-Dimensional Mesh Exchange Algorithm

It is worth noting that, just as the Ring Exchange algorithm is extended to the Mesh and Cube Exchange algorithms, it can also be extended to a similar algorithm exploiting a d-dimensional processor mesh for any $d \geq 1$. Moreover, using reasoning similar to that above, it is easy to show that the communications cost for this extended algorithm is

$$T_{d\text{-mesh}} \approx (P_{x_1} + P_{x_2} + \cdots + P_{x_d} - d)\left(t_s + \frac{MN}{2P} t_w\right) \tag{1.8}$$

where P_{x_i} is the number of processors in dimension i of the d-dimensional mesh with $P_{x_1} P_{x_2} \cdots P_{x_d} = P$ processors.

In total, each processor sends and receives $P_{x_1} + P_{x_2} + \cdots + P_{x_d} - d$ messages. This is optimal for a d-dimensional mesh with $P_{x_1} P_{x_2} \cdots P_{x_d} = P$ processors and uni-directional links, since it takes $P_{x_1} + P_{x_2} + \cdots + P_{x_d} - d$ hops to send a message from one corner of the mesh to the opposite corner. Moreover, this message count is asymptotically optimal for a mesh with bi-directional links.

If $P_{x_i} \approx \sqrt[d]{P}$ for $i = 1, \ldots, d$, then (1.8) reduces to

$$T_{d\text{-mesh}} \approx d(\sqrt[d]{P} - 1)\left(t_s + \frac{MN}{2P} t_w\right) \approx d\sqrt[d]{P} t_s + \frac{d}{2}\frac{MN}{P^{1-1/d}} t_w. \tag{1.9}$$

Moreover, for any $d \geq 1$, we can apply our d-dimensional mesh algorithm on any machine on which a d-dimensional mesh can be embedded. For example, it is well-known (Bertsekas89) that a d-dimensional mesh with $P_{x_i} = 2^{n_i}$ for $i = 1, \ldots, d$ can be embedded in an n-dimensional hypercube with $n = n_1 + \cdots + n_d$. On such systems, though, there may be other algorithms that better take advantage of the machine's high connectivity and thus give rise to lower communication costs. However, it is instructive to note that, if we apply our d-dimensional mesh algorithm with $P_{x_i} = 2$ for $i = 1, \ldots, d$ on a d-dimensional hypercube having $P = 2^d$ processors — thus making full use of the hypercube links — then (1.9) reduces to

$$T_{d\text{-mesh}} = d(\sqrt[d]{P} - 1)\left(t_s + \frac{MN}{2P} t_w\right) = d\left(t_s + \frac{MN}{2P} t_w\right)$$

since $\sqrt[d]{P} = 2$ in this case. This is the communication time for the standard hypercube data transposition algorithm (Foster95, p. 341). In fact, our d-dimensional mesh algorithm can be viewed as a generalization of the standard hypercube algorithm.

1.4 NUMERICAL EXPERIMENTS AND RESULTS

We used MPI (Gropp94) to implement the transpose schemes described in the previous section and incorporated these schemes into the shallow-water code developed by Steve Thomas at Environment Canada. This code is based on the numerical methods briefly described in §1.2. For more details, see (Ding98; Thomas94). Our numerical experiments were performed on both a Cray T3E machine with 512 processors and an ethernet-connected cluster of 36 Sun workstations.

The Cray T3E is a MIMD parallel machine and provides shared physical address space for up to 2048 processors over a 3D torus interconnection. The architecture of the machine can be characterized as *distributed shared-memory*: it has a global address space supported through remote get/put operations. Distributed memory programming interfaces, such as MPI and PVM, are supported. MPI for the Cray T3E was developed at the Edinburgh Parallel Computing Center in collaboration with Cray Research Inc.

The Computing Disciplines Facility (CDF) workstation cluster is a collection of Sun 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, a freely-available implementation of MPI, is used as the message passing environment on the CDF workstations.

Performance results from the parallel shallow-water code using different transposition algorithms are reported below. These algorithms are incorporated into the Fast Direct Solver for the linear 2D Helmholtz equations. The elapsed times for the transposition task, the linear 2D Helmholtz solver and shallow-water code are reported. Recall that transposition is performed twice within the linear 2D Helmholtz solver and that the linear 2D Helmholtz solver is called twice within each time-step of the shallow-water code. The elapsed times include computing time, communication time, idle time and all other overhead. To investigate the scalability of the algorithms, numerical experiments were performed for several problem sizes and numbers of processors. A theoretical scalability analysis of the algorithms is provided in (Ding98).

Figures 1.6–1.17 show graphically the parallel performance of the transposition algorithms, the linear 2D Helmholtz solver and the complete parallel shallow-water code for several grid sizes and numbers of processors on the Cray T3E. For each grid size, we plot the time required for the transpositions, the linear 2D Helmholtz solvers and the complete shallow-water code versus the number of processors on a log-log scale. For each grid size, we also plot the parallel efficiency of the shallow-water code versus the number of processors on a regular scale. In all figures, we denote by 'A' and 'D' our asynchronous and synchronous, respectively, implementations of the Direct Exchange algorithm, by 'R', 'M' and 'C' our implementations of the Ring, Mesh and Cube Exchange algorithms, and by 'L' our implementation of transposition using the MPI all-to-all communication routine `MPI_AlltoAll`.

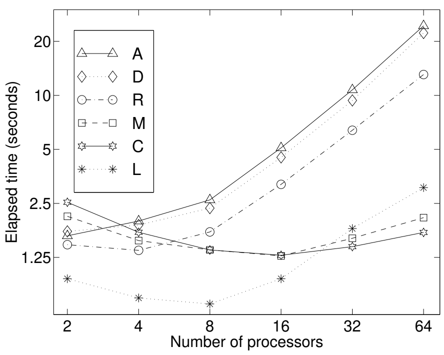

Figure 1.6 Performance of the transposition algorithms on the Cray T3E with 256×257 grid points.

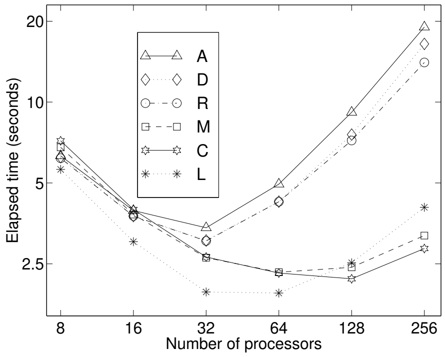

Figure 1.7 Performance of the Helmholtz solver using various transposition algorithms on the Cray T3E with 256×257 grid points.

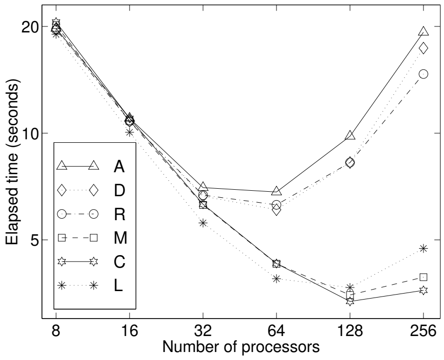

Figure 1.8 Performance of the shallow-water code using various transposition algorithms on the Cray T3E with 256×257 grid points.

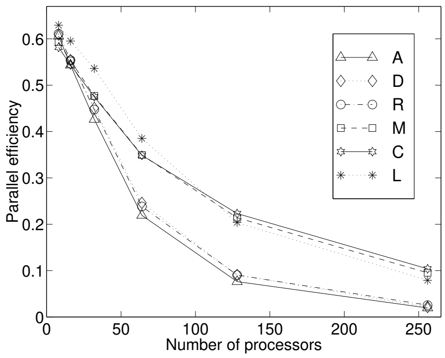

Figure 1.9 Parallel efficiency of the shallow-water code using various transposition algorithms on the Cray T3E with 256×257 grid points.

Figures 1.6, 1.10 and 1.14 show that the elapsed time for the Direct and Ring Exchange algorithms increases almost linearly asymptotically with P, as expected from relations (1.2) and (1.3). 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 and Ring Exchange algorithms. Moreover, the elapsed times for these faster schemes increase at a slower rate asymptotically. When P is small, the `MPI_AlltoAll` routine performs best. But when P is large, the Cube Exchange algorithm is fastest, followed by the Mesh Exchange algorithm. Asymptotically, the Cube Exchange algorithm scales the best, since it has the smallest rate of increase in elapsed time versus the number of processors.

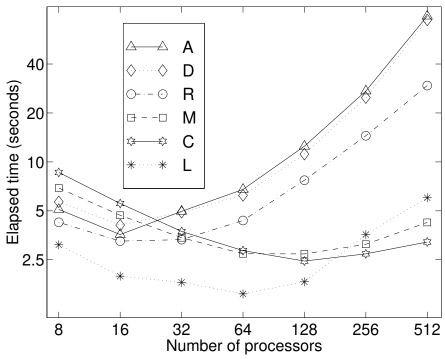

Figure 1.10 Performance of the transposition algorithms on the Cray T3E with 512×513 grid points.

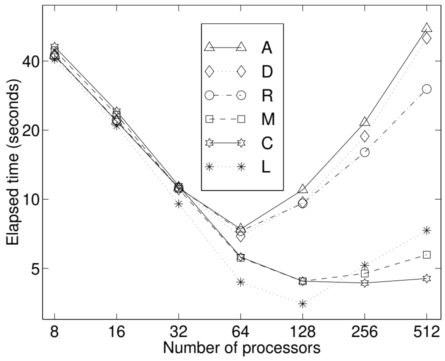

Figure 1.11 Performance of the Helmholtz solver using various transposition algorithms on the Cray T3E with 512×513 grid points.

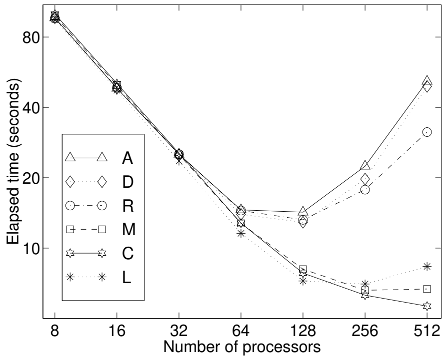

Figure 1.12 Performance of the shallow-water code using various transposition algorithms on the Cray T3E with 512×513 grid points.

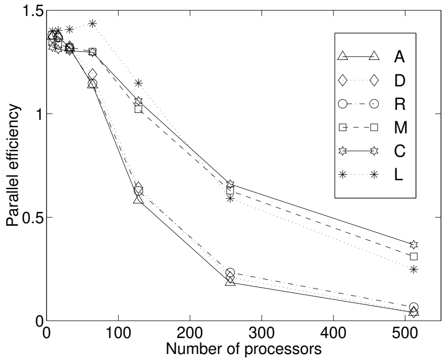

Figure 1.13 Parallel efficiency of the shallow-water code using various transposition algorithms on the Cray T3E with 512×513 grid points.

Note also that, for small P, the elapsed time for the Mesh and Cube Exchange algorithms, as well as the `MPI_AllToAll` routine, first decreases and then increases as P increases. This agrees with our analysis in §1.3 for the Mesh and Cube Exchange algorithms: equations (1.5) and (1.7) for T_{mesh} and T_{cube}, respectively, suggest that, for a fixed problem size of MN, the Mesh and Cube Exchange algorithms achieve a minimum elapsed time for $P \approx \frac{1}{2}MNt_w/t_s$ and $P \approx MNt_w/t_s$, respectively.

Figures 1.7, 1.11 and 1.15 show that the performance differences for the linear 2D Helmholtz solver with the various transposition algorithms are insignificant when P is small, since the computation time dominates. On the other hand, the performance differences are significant when P is large, since

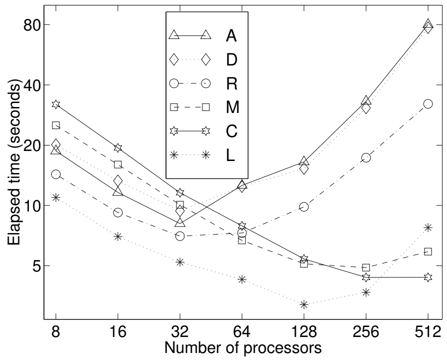

Figure 1.14 Performance of the transposition algorithms on the Cray T3E with 1024×1025 grid points.

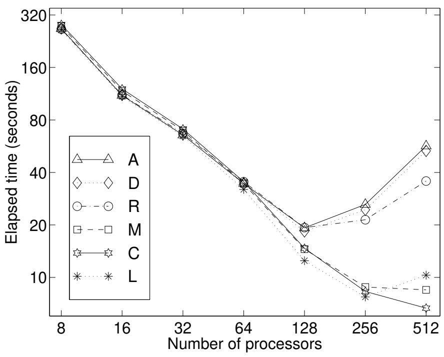

Figure 1.15 Performance of the Helmholtz solver using various transposition algorithms on the Cray T3E with 1024×1025 grid points.

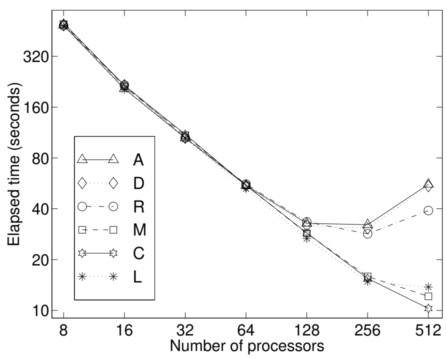

Figure 1.16 Performance of the shallow-water code using various transposition algorithms on the Cray T3E with 1024×1025 grid points.

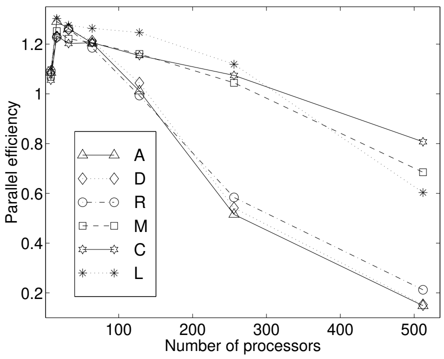

Figure 1.17 Parallel efficiency of the shallow-water code using various transposition algorithms on the Cray T3E with 1024×1025 grid points.

the communication time for the transposition schemes dominates. Figures 1.8, 1.12 and 1.16 show that the same is true for the complete shallow-water code.

Figures 1.9, 1.13 and 1.17 show that the parallel efficiency decreases as P increases. However, note that, as the grid size increases, the rate of efficiency loss decreases, especially for the more efficient Mesh and Cube Exchange algorithms. For the 1024×1025 grid, the efficiency for 512 processors remains above or close to 80% for these two schemes.

An interesting point is illustrated in Figures 1.13 and 1.17. For small P, the efficiencies are greater than 1. Moreover, for problem size 1024×1025, the efficiencies increase as P increases from 8 to 16. We believe the reason for this is the *cache effect*. The smaller P is, the more data must be allocated to each

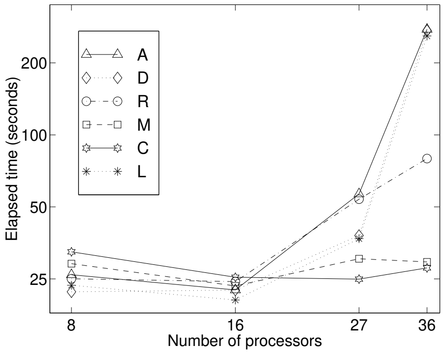

Figure 1.18 Performance of the transposition algorithms on the CDF system with 150×151 grid points.

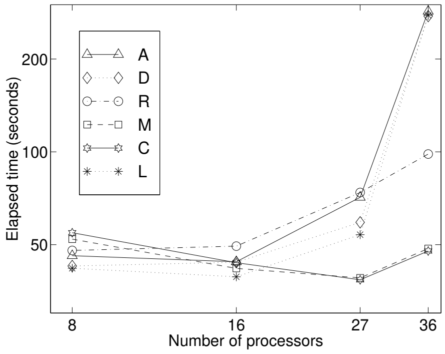

Figure 1.19 Performance of the Helmholtz solver using various transposition algorithms on the CDF system with 150×151 grid points.

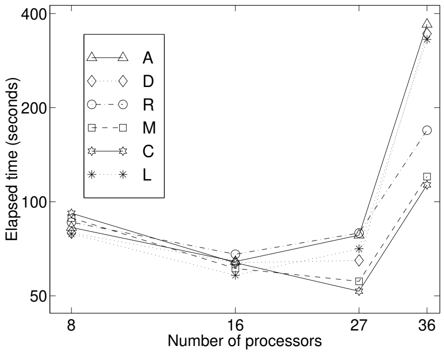

Figure 1.20 Performance of the shallow-water code using various transposition algorithms on the CDF system with 150×151 grid points.

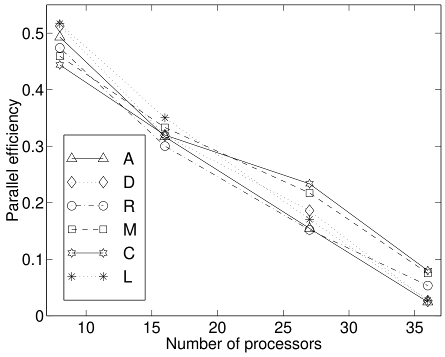

Figure 1.21 Parallel efficiency of the shallow-water code using various transposition algorithms on the CDF system with 150×151 grid points.

processor, the larger the block size m_b used in the transposition schemes, and the worse the utilization of the cache. Depending on the grid and cache sizes, cache misses may occur when manipulating local data or data in the message buffer. For a small number of processors, these cache effects are more costly than the communication overhead. The cache effect is reduced as the number of processors increases, and communication time becomes increasingly dominant.

Figures 1.18–1.21 show graphically the parallel performance of the transposition algorithms, the linear 2D Helmholtz solver and the complete parallel shallow-water code for a grid size of 150×151 and several numbers of workstations on the CDF system. The benefit from using the Mesh and Cube Exchange algorithms versus the `MPI_AlltoAll` routine is more dramatic on the CDF system than on the Cray T3E. This is because the ethernet-connected workstation cluster shares a single communication channel, and the many messages generated by `MPI_AlltoAll` create extensive contention. However, Figure 1.21 shows that the parallel efficiency of the complete shallow-water code is low in absolute terms when compared to that on the Cray T3E. Moreover, the parallel efficiency on the CDF system drops quickly as the number of processors increases. This is because (a) the grid size we were able to run is small and communication strongly dominates computation and (b) the sharing of the single communication channel by the processors causes some contention even when the Mesh and Cube Exchange algorithms are used.

1.5 DISCUSSION, CONCLUSIONS AND FURTHER RESEARCH

We have examined the data transposition requirements of a Fast Direct Solver for the linear algebraic equations that arise from the discretization of a linear 2D Helmholtz equation that must be solved at each time-step of a parallel shallow-water code. The transposition operation is found to be the bottleneck for the scalability of the shallow-water computation. We propose the Mesh and Cube Exchange algorithms that take advantage of the 2D and 3D topology, respectively, of the interprocessor communication network and require a smaller number of messages per processor than the commonly-used Direct and Ring Exchange schemes. No restriction is imposed on the dimensions of the matrix to be transposed, nor on the number of processors in each dimension. 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 shown that our new algorithms perform better asymptotically than the simpler Direct and Ring Exchange schemes as the number of available processors increases.

Our numerical results confirm this analysis. It is worth noting that the two parallel systems on which our experiments were run are representative of the two extremes of the spectrum of latency in message-passing systems: the Cray T3E has a low latency, while our ethernet-connected cluster of workstations has a high latency. In both cases, though, the Mesh and Cube Exchange algorithms outperform asymptotically the MPI all-to-all routine.

However, as shown in (Ding98), the parallel algorithm for the shallow-water equations remains essentially unscalable even with our more sophisticated transposition algorithms.

Further research can be done to improve the performance of the Mesh and Cube Exchange algorithms. Our schemes were implemented using high level MPI communication routines. Their performance may be improved significantly if more efficient low-level services were used.

In addition, we noted that the Mesh and Cube Exchange algorithms can be extended to a d-dimensional mesh of processors. Moreover, this extended algorithm can be applied on any machine on which a d-dimensional mesh can be embedded, for example, a hypercube. However, further work needs to be done to assess the effectiveness of the extended algorithm, particularly on machines, such as the SP2, that have an interprocessor connection network that is not directly related to a d-dimensional mesh.

Acknowledgments

We thank Steve Thomas, Ivar Lie and Roar Skålin for many helpful discussions.

References

D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, 1989.

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

X. Ding, *Numerical Solution of the Shallow-Water Equations on Distributed Memory Systems*, M.Sc. Thesis, Computer Science Department, University of Toronto, 1998. Available from `http://www.cs.toronto.edu/NA/reports.html`.

I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

W. Gropp, *Using MPI: portable parallel programming with the message-passing interface*, MIT Press, 1994.

S. L. Johnsson and C. T. Ho, *Algorithms for Matrix Transposition on Boolean N-Cube Configured Ensemble Architectures*, SIAM J. Matrix Anal. and Applic., vol. 9 (3), July 1988, pp. 419–454.

C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, 1992.

D. M. Nicol and S. H. Bokhari, *Optimal Multiphase Complete Exchange on Circuit-Switched Hypercube Architectures*, SIGMETRICS, ACM, vol. 22, no. 1, May 1994, pp. 252–260.

A. Robert, *A Semi-Lagrangian and Semi-Implicit Numerical Integration Scheme for the Primitive Meteorological Equations*, Jpn. Meteor. Soc., vol. 60, 1982, pp. 319–325.

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