INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
DATA ACCESS OPTIMIZATIONS FOR PARALLEL COMPUTERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

N.S. Sundar, M.S.

* * * * *

The Ohio State University

1998

Dissertation Committee:

Dr. P. Sadayappan, Adviser
Dr. Dhabaleshwar K. Panda
Dr. Mukesh Singhal

Approved by

Adviser
Department of Computer and Information Sciences
ABSTRACT

Parallel computers are increasingly used to deliver more computing power to demanding applications than is possible with uniprocessor systems. Another major trend affecting computer architecture is that processor speeds exceed memory speeds, with the differential increasing over time. This has led to the widespread use of cache memories to bridge the speed gap between processors and main memories.

Cache-based scalable parallel computers, whether with a unified logical memory address space or a distributed address space, are characterized by a higher latency for remote data access compared to local data access. Hence it is important to minimize the number and amount of off-processor memory accesses. Methods and techniques to maximize data locality are, therefore, of great importance in parallel computing.

This thesis explores issues in data locality in two different contexts. Firstly, on parallel systems with physically distributed memory, many existing applications need to perform collective communication to ensure data locality. Therefore, it is important to reduce the running time for these communication patterns. Complete exchange is an important pattern that arises in many commonly used applications. This study presents a new algorithm to perform complete exchange and shows that it can be effectively hybridized with existing algorithms to enhance performance.
Secondly, automatic conversion of sequential numerical programs to a parallel form often produces programs that perform poorly due to lack of data locality. Manual conversion to a (MPI-based) message-passing model addresses performance but is tedious and error-prone. So, it is important to create a parallelizing methodology that eases manual implementation without compromising performance. This study proposes a methodology for stencil computations that access their data with unit stride, which achieves ease of use by taking an incremental approach and by decoupling data distribution issues from communication issues. The resulting programs are demonstrated to scale well on many parallel architectures. Furthermore, on cache-based shared memory multiprocessors, the methodology enables an alternative to automatic parallelization and conversion to MPI, which can yield programs of performance comparable to MPI by exploiting data locality but takes less development effort. A performance study of the three approaches is presented to validate this conclusion.
This is dedicated to my beloved parents, Mrs. S. Kamala and Mr. N.S. Srinivasan.
I thank everyone who has contributed to this effort. In particular, I would like to thank Dr. P. Sadayappan whose consistent support and contribution made this endeavor possible. Dr. D.N. Jayasimha and Dr. Dhabaleshwar K. Panda provided me with valuable advice and guidance. Financial support from the Department of Computer and Information Science at The Ohio State University and from the Wright Patterson Air Force Base helped in the continuation of this study and I thank them for that. The parallel computing environments at the Ohio Supercomputer Center were invaluable in my work and the people at that organization were very prompt and helpful in resolving problems.
VITA

September 3, 1968 ......................... Born - Madras, India

1989 .............................................. B.Tech. Computer Science and Engineering

1991 .............................................. M.S. Computer and Information Sciences

1991-present ........................................ Graduate Teaching and Research Associate.
The Ohio State University.

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Sciences

Studies in:

- Parallel Computing  Prof. P. Sadayappan
- Computer Graphics  Prof. Richard Parent
- Operating Systems  Prof. Mukesh Singhal
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Complete Exchange</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Direct Exchange</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 The methodology of direct exchange</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 An analytical model for Direct Exchange</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Cyclic Exchange</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Cyclic Exchange algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Analytical models for Cyclic Exchange</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Hybridizing combining and direct algorithms</td>
<td>22</td>
</tr>
<tr>
<td>2.5 Simulation results and discussion</td>
<td>32</td>
</tr>
<tr>
<td>2.5.1 Results with hardware barrier</td>
<td>35</td>
</tr>
<tr>
<td>2.5.2 Results with software barriers</td>
<td>36</td>
</tr>
<tr>
<td>2.5.3 Results without barrier</td>
<td>39</td>
</tr>
</tbody>
</table>
2.6 Impact of contention on finish time .......................... 40
2.7 Influence of barriers on $K_{opt}$ ................................. 43
  2.7.1 Impact of hardware barriers ................................. 43
  2.7.2 Impact of software barriers ................................. 44
2.8 Experimental results ................................................. 46
2.9 Related Work .............................................................. 47
2.10 Conclusions .............................................................. 49

3. Incremental Parallelization of Stencil Computations ........ 51
  3.1 Introduction ............................................................. 51
  3.2 The Case Study .......................................................... 54
    3.2.1 Background ......................................................... 54
    3.2.2 Need for Manual Parallelization ......................... 55
    3.2.3 Elements of the Methodology .............................. 60
    3.2.4 Building a Blocked Program .............................. 61
    3.2.5 Building the MPI Program .............................. 66
  3.3 General Formulation of the Methodology .................... 68
    3.3.1 Domain of the Methodology ................................ 69
    3.3.2 Three Pass Structure ........................................... 70
    3.3.3 A Running Example ........................................... 71
    3.3.4 Data Distribution ............................................. 72
    3.3.5 The Blocking Pass ............................................. 81
    3.3.6 The MPI Pass .................................................. 97
  3.4 Performance Measurement .......................................... 99
  3.5 Related Work .......................................................... 103
  3.6 Conclusion .............................................................. 106

4. Alternatives for Parallelization in Shared Address Space Computers ... 108
  4.1 Introduction ............................................................. 108
  4.2 Automatic Parallelization: Pros and Cons .................. 110
  4.3 Generating the Blocked Parallel Program ................. 113
  4.4 Performance Evaluation of $fdl3di$ ......................... 115
  4.5 Conclusion .............................................................. 121

5. Contributions .............................................................. 123

Appendices:

ix
A. Data Movement in Cyclic Exchange .................................................. 126

B. Implementation issues in Cray T3D .............................................. 131

Bibliography ....................................................................................... 133
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Symbols and notation frequently used in this chapter.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Variation in completion times of CE and DE, and also in $K_{opt}$, as the parameters of complete exchange are varied one at a time. Among CE and DE, the one whose finish times increases the most is indicated with $\dagger$.</td>
<td>31</td>
</tr>
<tr>
<td>2.3</td>
<td>The actual (top row) and predicted (bottom row) completion times for various hybrids with hardware barriers, with barrier cost of $1\mu s$. The numbers in bold correspond to the best $K$ for that block size.</td>
<td>34</td>
</tr>
<tr>
<td>2.4</td>
<td>The $K$ of the best hybrid with hardware barriers, with barrier cost of $1\mu s$.</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>The $K$ of the best hybrid with software barriers.</td>
<td>38</td>
</tr>
<tr>
<td>2.6</td>
<td>The $K$-value of the best hybrid without barriers.</td>
<td>39</td>
</tr>
<tr>
<td>2.7</td>
<td>Comparison of completion times with and without hardware barriers for Direct Exchange and Cyclic Exchange on a $32 \times 32$ mesh.</td>
<td>40</td>
</tr>
<tr>
<td>2.8</td>
<td>The best $K$ and hardware barrier usage for various $p$, $m$ and $t_s/t_p$. The letter “h” denotes the use of hardware barriers while “n” indicates absence of barriers.</td>
<td>44</td>
</tr>
<tr>
<td>2.9</td>
<td>The best $K$ and software barrier usage for various $p$, $m$ and $t_s/t_p$. The letter “s” denotes the use of software barriers while “n” indicates absence of barriers.</td>
<td>45</td>
</tr>
<tr>
<td>2.10</td>
<td>The $K$ of the best performing hybrid on the Cray T3D. Note that $8 \times 8$ mesh is embedded in a $8 \times 4 \times 4$ torus.</td>
<td>47</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Scott’s Direct exchange on a 1D chain of processors.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A cross product of two 1D patterns in Scott’s Direct Exchange.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Cyclic Exchange on a linear array of 8 nodes.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Cyclic Exchange on a 4×4 mesh. if the processors execute in lockstep.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>The Cyclic Exchange Algorithm for an 8×8 mesh.</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>The Cyclic Exchange Algorithm for Complete Exchange.</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>The topmost row of a 16×16 mesh at the time when the last flit in the horizontal message of ( W_1 ) has reached its destination. The black node ( B_1 ) has also received its last flit and is performing a startup. It cannot block ( W_2 ) but, if the startup time were small, it can block ( W_1 ), causing multicolor contention.</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>The decision tree for switching to Direct Exchange in a hybrid scheme.</td>
<td>23</td>
</tr>
<tr>
<td>2.9</td>
<td>Variation of contention factor with block size for Cyclic Exchange and Direct Exchange in the absence of barriers for a 32×32 mesh. Contention factor is defined as the ratio of average actual latency of all messages in the algorithm to the average ideal (contention-free) latency.</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>A 2D skewed distribution, indicating the processors that own the tiles.</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>A running example implementing an equation solver using Jacobi.</td>
<td>72</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm to convert a given array reference to a blocked form.</td>
<td>76</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Parallel computers are increasingly used to deliver more computing power to demanding applications than is possible with uniprocessor systems. Another major trend affecting computer architecture is that processor speeds exceed memory speeds, with the differential increasing over time. This has led to the widespread use of cache memories to bridge the speed gap between processors and main memories.

Cache-based scalable parallel computers, whether with a unified logical memory address space or a distributed address space, are characterized by a higher latency for remote data access compared to local data access. Hence it is important to minimize the number and amount of off-processor memory accesses. Methods and techniques to maximize data locality are, therefore, of great importance in parallel computing.

This thesis explores issues in data locality in two different contexts. Firstly, on parallel systems with physically distributed memory, many existing applications need to perform collective communication to ensure data locality. Therefore, it is important to reduce the running time for these communication patterns. Complete exchange is an important pattern that arises in many commonly used applications. This study presents a new algorithm to perform complete exchange and shows that it can be effectively hybridized with existing algorithms to enhance performance.
Secondly, automatic conversion of sequential numerical programs to a parallel form often produces programs that perform poorly due to lack of data locality. Manual conversion to a (MPI-based) message-passing model addresses performance but is tedious and error-prone. So, it is important to create a parallelizing methodology that eases manual implementation without compromising performance. This study proposes a methodology for stencil computations that access their data with unit stride, which achieves ease of use by taking an incremental approach and by decoupling data distribution issues from communication issues. The resulting programs are demonstrated to scale well on many parallel architectures. Furthermore, on cache-based shared memory multiprocessors, the methodology enables an alternative to automatic parallelization and conversion to MPI, which can yield programs of performance comparable to MPI by exploiting data locality but takes less development effort. A performance study of the three approaches is presented to validate this conclusion.

The dissertation is structured as follows. Chapter 2 presents Cyclic Exchange and demonstrates that hybridizing it with existing algorithms to enhance performance. Chapter 3 develops an incremental methodology to parallelize stencil computations that access their data with unit stride, which is well-suited to manual implementation. Chapter 4 presents a new alternative to parallelize such stencil computations which can be implemented with less effort than conversion to MPI and yet yields comparable performance. Chapter 5 summarizes the contributions made by this thesis.
CHAPTER 2

COMPLETE EXCHANGE

2.1 Introduction

The all-to-all personalized (complete) exchange is a global collective communication operation in distributed memory multicomputers in which every processor sends a unique block of data to every other processor in the system. This pattern arises in many important problems such as multidimensional FFT, matrix transposition and sorting. It is one of the collective communication primitives defined by the MPI [15] message passing interface standard. Since complete exchange is communication intensive, its effective implementation in wire-sparse topologies like 2D meshes is a challenging problem.

The principal issues that need to be addressed in an effective solution to the complete exchange problem are those of high message startup overhead, the high volume of data that needs to be transmitted over the network and contention among messages for the mesh links. If every processor in an \( n \)-processor mesh directly delivers its block of data to every other processor in the system, each processor needs to execute at least \( n - 1 \) message startups sequentially. As many modern multicomputers have a high message startup time, which is often orders of magnitude higher than
the time to transmit a byte over a link. This factor strongly influences the completion time.

The total volume of data that the network has to carry is at least \( n(n - 1)m \) bytes in an \( n \)-processor system, where \( m \) is the size of each data block. Since this is quadratic in the system size, it is another significant factor that affects performance. In a 2D mesh, the limited bisection bandwidth implies that the central links should be optimally used to achieve good performance.

Furthermore, contention is a significant factor in wormhole-routed networks because wormhole routing reserves network links. When a message is forced to wait for a link to become available, the links which it holds are not released, which can cause other messages to block. So, significant network bandwidth may become unusable and message latency increases. Thus, contention is a critical factor that needs to be managed.

Past approaches to performing complete exchange on 2D meshes have taken one of two distinct approaches. The direct exchange approach involves each processor directly sending a block of data to every other processor in the system. For example, the Optimal algorithm for hypercubes [24], in which every processor exchanges data with processor \( myid \oplus i \) in step \( i \), can be adapted for meshes. Thakur et al. [44, 43] evaluate algorithms that take this approach. Scott [36] has proven that, in an \( a \times a \) mesh, the number of contention-free communication steps, each of which overlaps messages between several processor pairs, must be at least \( \frac{1}{4}a^3 \). He also presents a methodology for developing direct exchange algorithms that realize this lower bound when \( a \) is a multiple of 4.
In contrast to direct exchange, an alternative approach is to combine the data blocks that a processor $P$ has for all processors in a submesh and send them as a single message to one processor $Q$ within that submesh. $Q$ then forwards the data blocks towards their respective destinations in subsequent phases. This combining or indirect approach typically takes a divide-and-conquer strategy. For example, Bokhari and Berryman [10] developed two algorithms that use this approach called Binary Exchange and Quadrant Exchange. Binary Exchange recursively divides the mesh into halves while Quadrant Exchange organizes the mesh in quadrants. The latter has been shown to be faster than the former. Sundar et al. [41] proposed another combining algorithm called Cyclic Exchange, which recursively divides the mesh into quadrants. (Such an algorithm was also proposed independently by Gupta et al. [21].) This was demonstrated to be faster than Quadrant Exchange for a wide range of mesh sizes and data block sizes.

The direct exchange and the indirect (combining) approach have complementary strengths. Direct Exchange minimizes the volume of data that must be transmitted sequentially over the network but has a large number of message startups. Combining approaches reduce the number of startups considerably, but move larger volumes of data over the network since some blocks are retransmitted several times. This tradeoff suggests that both these schemes may be integrated to exploit their complementary strengths.

This chapter presents a family of algorithms for complete exchange on wormhole-routed 2D meshes, derived by hybridizing Cyclic Exchange and Scott’s Direct Exchange. These hybrids are formulated by replacing the last $K$ phases in the recursive structure of Cyclic Exchange with Direct Exchange. The value of $K$ that minimizes
completion time will depend on system parameters (mesh size, message startup time, flit transfer time and barrier cost) and the problem parameter (data block size). Analytical models are presented for the hybrids, which may be used to pick the best hybrid for a given combination of parameters when barriers are used (see below). The analytical models are validated with a simulation study and an implementation on the Cray T3D multicomputer, whose results confirm that the hybrids are the fastest algorithms for a large range of system and problem parameters. A unique feature of the study is that link contention is quantified accurately with an analytical model, unlike many past studies which model contention in an approximate manner. The chapter also investigates the use of barriers, implemented either in hardware or software, to reduce contention, thereby improving performance. A counter-intuitive experimental observation is presented and explained, where higher message startup costs actually reduce completion time.

The chapter is organized as follows. Section 2.2 describes Direct Exchange [36] and presents an analytical model which is applicable when the algorithmic steps execute in synchrony (as in an SIMD architecture or in an MIMD architecture with barrier synchronization between steps). Section 2.3 presents Cyclic Exchange and presents analytical models for it, both with and without the assumption of synchrony. The motivation for hybridizing and the structure of the hybrid algorithms are discussed in Section 2.4, which also uses the analytical model to prove theorems that provide useful insights into the behavior of the hybrids. Section 2.5 presents the results of a simulation study of hybrid algorithms, with and without the use of barriers. The simulation results are in close correspondence to the analytical predictions and show that the proposed hybridizing scheme is effective for a wide range of system and
problem parameters. Section 2.6 discusses the influence that message contention has on finish time and presents evidence that use of barriers can enhance performance by reducing contention. Since the target system may or may not provide hardware support for barrier synchronization, section 2.7 focuses on the effectiveness of barrier usage in both cases. The results of the implementation of the hybrid algorithms on the Cray T3D multicomputer are presented in Section 2.8 and they are shown to confirm the trends seen with the analytical model and the simulation study. Section 2.9 points to previous related literature and shows how this work differs from them. Finally, Section 2.10 summarizes the work.

Table 2.1 explains the notation commonly used in the rest of the chapter.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>Cyclic Exchange</td>
<td>DE</td>
<td>Direct Exchange</td>
</tr>
<tr>
<td>$m$</td>
<td>size (in bytes) of a data block</td>
<td>$M$</td>
<td>size (in bytes) of a message in CE</td>
</tr>
<tr>
<td>$b$</td>
<td>Cost of barrier synchronization</td>
<td>$t_{mem}$</td>
<td>time to copy a byte within local memory</td>
</tr>
<tr>
<td>$t_p$</td>
<td>Time for a flit to cross a mesh link</td>
<td>$t_s$</td>
<td>Message startup overhead</td>
</tr>
<tr>
<td>$K$</td>
<td>the number of steps in a hybrid algorithm that are carried out with Direct Exchange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p$</td>
<td>logarithm of the mesh size i.e. mesh size is $2^p \times 2^p$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Symbols and notation frequently used in this chapter.

2.2 Direct Exchange

There are several algorithms [36, 44, 43] for complete exchange that schedule a direct communication between every pair of processors. The main issues in this approach are the large number of message startups and contention for the central links of the mesh, which form a bottleneck. In an $a \times a$ mesh, a lower bound can be derived for the number of contention-free communication steps needed, by noting
that there are $\frac{a^2}{2}$ nodes on either half of the mesh, so that $\frac{a^4}{4}$ data blocks need to cross the central links, but there are only $a$ central links: therefore there need to be at least $\frac{a^3}{4}$ sequential steps.

2.2.1 The methodology of direct exchange

Scott [36] presented a scheme to generate complete exchange algorithms which achieve the above lower bound when $a$ is a multiple of 4. The central idea behind the scheme is a set of contention-free communication patterns for complete exchange on a one-dimensional chain of processors, which is then extended to two dimensions by the use of cross-products of these patterns in a contention-free manner. The following is a description of this methodology.

On a chain of $a$ processors, numbered 0, 1, ..., $a - 1$, define the antipodal node of processor $x$ as $(a - 1) - x$. Designate two nodes in the left half of the chain as A and B. Let their corresponding antipodal nodes be C and D, respectively. A communication pattern in which A sends to B, B to C, C to D and D to A, would utilize the central link in both directions (see Figure 2.1). By varying A and B over the $\frac{a}{2}$ nodes in the left half of the mesh (such that they are always distinct), $\frac{a}{2}(\frac{a}{2} - 1)$ regular steps are generated, which collectively do a complete exchange except for communication.
among the antipodal nodes themselves. The latter is achieved through a set of \( \frac{a}{2} \) *special* steps, in each of which a node on the left side exchanges a data block with its antipodal node on the right side. Thus, a total of \( \frac{a^2}{4} \) steps are needed (if \( a \) were even).

Each set of 4 processors in the 2D mesh shaded as □ □ □ □ and □ communicate independently with each other. Processors which are shaded □ are not involved in the cross product.

![Cross Product of Two 1D Patterns](image)

Figure 2.2: A cross product of two 1D patterns in Scott's Direct Exchange.

On a 2D mesh, complete exchange can be carried out by executing cross products of all the (regular and special) steps in the vertical and horizontal dimension. Figure 2.2 shows one such cross product. To include communication among processors in the same row (or the same column), it is necessary to assume that, in the special steps, there are two processors that are "actively sending to themselves". If every cross product were executed sequentially, this would result in \( \frac{a^4}{16} \) steps, which is suboptimal compared to the lower bound of \( \frac{a^3}{4} \). The key to deriving an optimal scheme is to observe that, while in a 4 × 4 mesh, all the processors are busy in every step, that is not the case with \( 4k \times 4k \) meshes for \( k > 1 \). In fact, the 1D schemes would involve only
four processors in each row (and in each column), so that \( k \) separate cross products can be scheduled within a step, as long as they don't contend among one another. Scott presents a method based on the tournament scheduling problem to choose the \( k \) cross products that constitute each step. Thus, the number of steps is reduced by a factor of \( k = a/4 \), so that the total number of steps is optimal.

In the following, any of the 2D algorithms derived by this methodology will be called Direct Exchange.

### 2.2.2 An analytical model for Direct Exchange

Each step in Direct Exchange has no link contention. However, when they are executed on an MIMD architecture, the steps are not synchronous, so that link contention will result. This is because not all processors will finish a given step together, even if they started it together. Besides, some processors are idle in steps that involve the special phases, so that they will attempt to begin the next step and thus contend with processors executing the current step.

Contention is difficult to quantify and model in Direct Exchange, in part because it depends on the sequence of cross products that get scheduled. Furthermore, our simulation results (presented in Section 2.5) show that controlling the contention by imposing a barrier between successive steps (so that the steps execute synchronously) results in better completion time, for a large range of mesh sizes and problem sizes. Hence, an analytical model is presented under the assumption that the steps are executed synchronously, with a barrier between successive steps.
On an \( a \times a \) mesh, there are \( \frac{a^2}{4} \) steps. Each step incurs a message startup cost and a message transmission cost. The time for a message transmission in a wormhole-routed mesh has two components: one for the header to establish a path (\( t_h \)) and one for the rest of the message to follow in pipeline fashion, the latter being typically dominant. The first term depends on the number of hops the message needs to travel. Thus, the header terms for the longest messages within each step can be determined, once the schedule of cross products is fixed, and their sum \( T_h \) can be found. This yields the completion time

\[
T_{DE} = \frac{a^3}{4} t_s + \frac{a^3}{4} m t_p + T_h + \left( \frac{a^3}{4} - 1 \right) b
\]  

(2.1)

The cost due to startups is quite high, though the transmission time component is optimal.

### 2.3 Cyclic Exchange

In contrast to the direct exchange approach, the combining approach gathers several data blocks that a given processor \( P \) has for an entire submesh and sends them in a single message to a processor \( Q \) within that submesh. \( Q \) then forwards the appropriate data blocks to the various submeshes within that submesh. This gives a recursive structure to the combining algorithms known today.

The Cyclic Exchange (CE) algorithm [41] works on a 2D circuit-switched or wormhole-routed mesh. The algorithm is described for a \( 2^p \times 2^p \) mesh but it can be adapted to non-square meshes with non-power-of-two number of processors to each side.
2.3.1 Cyclic Exchange algorithm

We first present the underlying idea for a one-dimensional array of $2^p$ nodes. Figure 2.3 illustrates it for $p = 3$. The communication occurs in $p$ phases such that, in phase $i$ ($i = p-1, \ldots, 0$), processors that are $2^i$ hops away exchange data between themselves. Consider a system with $p = 3$, that is $2^3$ processors. In the first phase, every processor communicates with its counterpart (4 hops away) in the other half of the linear array. In the second phase, each processor exchanges with its counterpart (2 hops away) in the other quarter of the same half. The last phase finally swaps data among neighboring nodes. The size of the messages remains constant at $4m$ bytes in each phase, $m$ being the size of a data block.

![Diagram of Cyclic Exchange on a linear array of 8 nodes.](image)

Figure 2.3: Cyclic Exchange on a linear array of 8 nodes.

For a 2D mesh, CE interleaves this pattern in both dimensions. Assume initially that the communication is organized in subphases that execute synchronously. The algorithm for two dimensions is illustrated in Fig. 2.4 for a $4 \times 4$ mesh. In the initial phase, the entire mesh is involved and the messages are of length $2^{2p-1}m$ bytes. Subsequent phases recursively divide the mesh into quadrants and operate on each
Figure 2.4: Cyclic Exchange on a $4 \times 4$ mesh, if the processors execute in lockstep.

quadrant independently. Phase $i$ (where $i = 1, 0$) operates on a $2^{i+1} \times 2^{i+1}$ submesh and has $2^i$ subphases because that many processors have to communicate across the central link in each row (except for phase $i = 0$ which has two subphases). Phase $i = p - 1$ is performed first.

Figure 2.5: The Cyclic Exchange Algorithm for an $8 \times 8$ mesh.
In an MIMD system, the subphases will not execute synchronously. Different communications will get overlapped, both within and across phases. The algorithm behavior under these conditions is illustrated in Fig. 2.5 with an $8 \times 8$ mesh. Three phases are required: during phase $i$ (where $i = 2, 1, 0$), each processor communicates with two others that are $2^i$ hops away, one in the same column and one in the same row. Thus, communications occur independently in submeshes of size $2^{i-1} \times 2^{i-1}$. A checkerboard pattern is used to interleave communication so that, in each phase, in the first step, half the processors communicate horizontally (the white nodes) and the other half communicate vertically (the black nodes). In the next step, white nodes communicate vertically and black nodes communicate horizontally. For example, in Figure 2.5, processors marked with a white circle first communicate horizontally to the similarly marked processor in the same row, and then communicate vertically to the similarly marked processor in the same column.

Every processor has an array of $2^p$ data blocks. In every communication, two data quadrants of the data array are sent to the partner node, so that message size is constant throughout the algorithm at $2^{2p-1}m$ bytes. The data need to be organized appropriately so that, in each communication, the set of data blocks to be sent are contiguous in the data array. This necessitates two kinds of movement of data blocks: reorder between phases and data movement within each phase. Our implementation of Cyclic Exchange uses a scheme that overlaps all intraphase data movement and half the reordering with communication, by using a user buffer which is $3.2^{2p-2}m$ bytes in size. Details of this scheme are provided in Appendix A, along with the Cyclic Exchange algorithm. A high-level outline of the algorithm showing the major communication steps and the data movement is presented in Figure 2.6.
\{ SPMD program executed by the processor in row \( r \) and column \( c \). \}

**algorithm** CyclicExchange\((r, c)\).

\[\text{begin}\]
\[\text{for} \ i \leftarrow p - 1 \ \text{downto} \ 0 \ \text{do} \ \{\ \text{Phase} \ i \ \}\]
\[\text{HorizProc} \leftarrow (r, c \pm 2^i)\]
\[\text{VertProc} \leftarrow (r \pm 2^i, c)\]
\[\text{if} \ (r \mod 2 = c \mod 2) \ \text{or} \ (i = 0) \ \{\text{White Node}\}\]
\[\text{Partner1} \leftarrow \text{HorizProc}; \text{Partner2} \leftarrow \text{VertProc}\]
\[\text{else} \ \{\text{Black Node}\}\]
\[\text{Partner2} \leftarrow \text{HorizProc}; \text{Partner1} \leftarrow \text{VertProc}\]
\[\text{endif}\]
\[\text{Step1: Exchange} \ 2^{2p-1} \ \text{data blocks with Partner1}\]
\[\quad \text{Perform intra-phase data movement of} \ 2^{2p-2m} \ \text{bytes}\]
\[\text{Step2: Exchange} \ 2^{2p-1} \ \text{data blocks with Partner2}\]
\[\quad \text{if} \ (i > 0) \ \text{perform interphase data reordering of} \ 2^{2p-1m} \ \text{bytes}\]
\[\text{endfor}\]
\[\text{end}\]

\[\text{Figure 2.6: The Cyclic Exchange Algorithm for Complete Exchange.}\]
2.3.2 Analytical models for Cyclic Exchange

In an MIMD implementation of Cyclic Exchange, in general, messages will contend for mesh links. The pattern of contention among messages can be predicted and modeled. However, when mesh size $p$ and data block size $m$ exceed certain limits, contention becomes high and difficult to model analytically. In fact, as our simulation results show (Section 2.5), the contention may be so high in this parameter range that imposing a barrier between successive steps or phases decreases completion time by eliminating contention among messages that belong to different steps/phases. When barriers are imposed between phases, it is feasible to analytically model the algorithm's behavior for those parameter ranges. Accordingly, we use a three-part model: one part without barriers which is valid for small mesh size $p$ and data block size $m$ and two others with barriers between phases or steps which is valid for large $p$ and large $m$. A guideline for deciding which part of the model to use is also formulated (see Condition 2.4 in this section).

In the absence of barriers between steps, there are two kinds of message contention in general in Cyclic Exchange. One is contention among messages originating from nodes of the same color. For example, in any phase, all the white nodes on the left side of any given row will send a message each to their partners on the right side; all these will contend for the central link of the row. This phenomenon is called same-color contention. Therefore, in phase $j$, which happens on a $2^{j+1} \times 2^{j+1}$ submesh, a message from the far left white node of a row will reach its destination with a latency of

$$T_{sc}^{(j)} = t_p^{(j)} + 2^{j-1} M t_p \quad (2.2)$$
where \( t^{(j)}_h \) is the time for header movement for one message (since the header movements of other white messages are overlapped with data transmissions) and \( M = 2^{2^{p-1}}m \) is the message size when data blocks are \( m \) bytes long.

Figure 2.7: The topmost row of a 16 \( \times \) 16 mesh at the time when the last hit in the horizontal message of \( W_i \) has reached its destination. The black node \( B_1 \) has also received its last flit and is performing a startup. It cannot block \( W_2 \) but, if the startup time were small, it can block \( W_4 \), causing multicolor contention.

The other kind of message contention is contention among messages from nodes of differing colors. For example, the message from a white node (call it a white message) on the far left of a given row may be so delayed by other white messages from the same half of the row that a black node in that half may finish the reception of its first (vertical) message, perform the message startup for the next (horizontal) communication and initiate the horizontal communication, thus colliding with the white message and delaying it further. In Figure 2.7 which illustrates this situation, the last flit from the white node labeled \( W_i \) has reached its destination, and simultaneously the black node \( B_1 \) has received the last flit of its first (vertical) message. The next (horizontal) message of \( B_1 \) cannot block \( W_2 \) as the latter's message has followed that of \( W_i \) in pipeline fashion. However, it can block the other white nodes depending on how long the intervening message startup is. Since all the black nodes in this row receive their vertical messages at the same time, all of them can contend with and
delay \( W_4 \), except for the immediately adjacent black node \( B_4 \). Of course, the other white nodes will also delay \( W_4 \) by same-color contention. Thus, the time when the partner of \( W_4 \) receives its message is

\[
T_{mc}^{(j)} = t_h^{(j)} + (2^i - 1) M t_p
\]

(2.3)

Note that, in an \( 8 \times 8 \) mesh, multicolor contention cannot happen because there are only two white nodes on each half of a row. Therefore, it cannot happen on smaller meshes either. Similarly, if barriers were imposed between successive steps in each phase, multicolor contention would be eliminated. On a \( 16 \times 16 \) mesh, whether a black node can delay a white node in the same half depends on the relative magnitudes of the message startup time \( t_s \) and the time to transmit one message \( M t_p \). If

\[
t_s < M t_p \text{ and } p \geq 4.
\]

(2.4)

multicolor contention can be expected to be significant. Since this is the case for \( 32 \times 32 \) and larger meshes for practically any value of \( m \). for reasonable values of \( t_s \), and \( t_p \), it is better to impose a barrier for large \( p \).

The remainder of this section presents analytical models for Cyclic Exchange for three cases: barrier synchronization between steps (and between phases too) , barrier synchronization between phases only and no barrier synchronization at all.

**Model with barrier between steps**

This model is applicable when successive steps have a barrier between them, so that multicolor contention is eliminated. Within each step, the message startups are overlapped. Thus, with \( 2p \) steps. the total startup time is

\[
S = 2p t_s
\]
In phases that involve $8 \times 8$ or larger meshes, messages from nodes of the same color will contend for central links. For example, horizontal messages from white nodes on one half of a row will all contend for the central link of that row. Thus, each step in phase $j$ will involve a time of $2^{j-1}M_{tp}$. The phases involving $4 \times 4$ and smaller meshes have no such contention. Thus, the sum of the non-overlapped transmission times is

$$\sum_{j=1}^{p-1} 2.2^{j-1}M_{tp} + 2M_{tp}$$

ignoring the times for header movement in each message. This evaluates to

$$T = 2^{3p-1}mt_p$$

Since only half the data array is reordered between successive phases, the total time spent in reordering is

$$R = (p - 1)2^{2p-1}mt_{mem}$$

The fourth component of the completion time is the barrier overhead of

$$B = (2p - 1)b$$

So, the completion time can be expressed as the sum of these four components $S + T + R + B$. that is

$$T_{CE}^{barr} = 2pt_s + 2^{3p-1}mt_p + (p - 1)2^{2p-1}mt_{mem} + (2p - 1)b$$

(2.5)

Model with barrier between phases

If the condition 2.4 holds true, multicolor contention becomes significant and causes a large variation in the times when different processors finish each phase. This leads to considerable contention among messages belonging to successive phases. thus
affecting the completion time. Interphase message contention can be eliminated by imposing a barrier between successive phases. This results in an additional cost of $p-1$ barriers, but if the barrier cost were low, the reduction in contention can decrease the completion time.

Each phase can be analyzed independently in the presence of barriers, as all processors begin a phase together. In $16 \times 16$ and larger meshes, multicolor contention will delay the message reception of processors such as $W$ (see figure 2.7) considerably, so that the highest latency is $T_{mc}^{(j)}$ in phase $j$. However, it is only the nodes farthest from the center of the mesh that will face this high degree of contention: others will finish their message reception sooner. Therefore, in the next step of the same phase, far fewer processors will be involved in contention. As the exact number $N_j$ of contending processors will depend on the dynamics of flit movement in the previous communication, it is difficult to give a general closed-form formula to estimate it. However, for specific mesh sizes. $N_j$ can be determined. For example, for a $16 \times 16$ mesh in phase 3. $N_3 = 2$ and for a $32 \times 32$ mesh in phase 4. $N_4 = 5$. Thus, the time for phase $j$ can be written as

$$T_j = t_s + reorder_j + \left\{ \begin{array}{ll} T_{mc}^{(j)} + N_j & \text{if } j \geq 3 \\ \frac{T_{mc}^{(j)}}{2T_{sc}^{(j)}} & \text{else} \end{array} \right\}$$

The completion time is the sum of all phase times, including the barrier cost is

$$\sum_{j=0}^{j=p-1} T_j + (p - 1)b$$

Model with no barriers

In the absence of barriers, all processors execute asynchronously so that different phases may overlap. The model for this case has to consider the critical path i.e. the
sequence of message transmissions that take the longest to complete. Accordingly, for each phase \( j \) \((p - 1 \geq j \geq 0)\), define \( \text{earliest}_j \) as the time when phase \( j \) can be completed earliest, and \( \text{latest}_j \) as the time when phase \( j \) is completed by the last processor(s) (i.e. when both the horizontal and vertical messages have been received and reordering has been performed). Recurrence equations can be defined for both these sets of times. The time \( \text{latest}_0 \) gives the completion time for the algorithm.

To derive the recurrence equation for \( \text{earliest}_j \), note that the processors in the initial phase \( j = p - 1 \) that face no contention in message reception in either dimension will finish first. In the last phase \( j = 0 \), all messages occur within \( 2 \times 2 \) meshes, so that there is no contention here either. In all other phases, the processors that finished earliest in previous phases will be able to initiate a transmission first but will face same-color contention. Thus,

\[
\text{earliest}_j = 2t_s + \text{reorder}_j + \begin{cases} 
\text{earliest}_{j+1} + 2 \times T^{(j)}_{\text{free}} & \text{if } j = p - 1 \text{ or } j = 0 \\
\text{earliest}_{j+1} + 2 \times T^{(j)}_{\text{cont}} & \text{else}
\end{cases}
\]

(2.6)

for \( p - 1 \geq j \geq 0 \). Here \( T^{(j)}_{\text{free}} = t_h^{(j)} + M_{tp} \) is the latency for a contention-free transmission and \( T^{(j)}_{\text{cont}} \) is either \( T^{(j)}_{sc} \) or \( T^{(j)}_{mc} \) depending on condition 2.4. The reorder term is \( 2^{p-1}MT_{\text{mem}} \) only if \( j > 0 \). In the last phase \( j = 0 \), it is zero.

The last processor to finish a phase would have faced contention in receiving messages in either dimension, unless it were the \( 2 \times 2 \) submesh phase, or if it were receiving from a processor that finished the previous phase so late that, in the current phase, it faced no contention. Thus,

\[
\text{latest}_j = 2t_s + \text{reorder}_j + \begin{cases} 
\text{latest}_{j+1} + 2 \times T^{(j)}_{\text{free}} & \text{if } \text{latest}_{j+1} > \text{earliest}_{j+1} + T^{(j)}_{\text{cont}} \\
\text{latest}_{j+1} + 2 \times T^{(j)}_{\text{free}} & \text{if } j = 0 \\
\text{latest}_{j+1} + 2 \times T^{(j)}_{\text{cont}} & \text{else}
\end{cases}
\]

(2.7)

The reorder term is defined in the same way as above.
2.4 Hybridizing combining and direct algorithms

To motivate the hybridizing scheme, consider the behavior of Direct Exchange (DE) and Cyclic Exchange (CE), both with barriers between steps. The completion time for DE for a $2^p \times 2^p$ mesh can be expressed as (Equation 2.1):

$$2^{3p-2}t_s + 2^{3p-2}mt_p + (2^{3p-2} - 1)b$$

The completion time for CE is (Equation 2.5)

$$2pt_s + 2^{3p-1}mt_p + (p - 1)2^{2p-1}mt_{mem} + (2p - 1)b$$

Thus, the startup component is much higher with DE but the transmission time component is optimal. With CE, the startup component is much lower but its transmission volume is double that of DE and it also incurs a reordering overhead. This suggests that both the schemes can be combined to exploit their complementary benefits.

Consider an execution of CE on a $2^p \times 2^p$ mesh, with $p$ phases. In phase $j$, where $p - 1 \geq j \geq 0$, CE operates on independent submeshes of size $2^{j+1} \times 2^{j-1}$. The communication performed in a $2^{j+1} \times 2^{j-1}$ submesh for phases $j$ and lower is equivalent to performing complete exchange on a mesh of that size with an effective data block size of $2^{2(p-1-j)}m$. Thus, in the recursive structure of CE, successive phases operate on smaller submeshes with increasing effective block sizes. A decrease in the mesh size implies that startup overhead becomes less significant. The increased block size implies a higher transmission volume. Both these factors favor DE. So, at some point in the sequence, it may become more advantageous to perform a single direct exchange within each submesh instead of carrying out the remaining phases of CE.
That is, at some phase \( j = K - 1 \), it may be beneficial to change to DE and replace the last \( K \) phases of CE with a single DE operation, which is executed in every \( 2^K \times 2^K \) submesh independently. The effective block size in the DE stage of such a hybrid algorithm is \( 2^{2p-2K}m \).

Thus, a family of \( p+1 \) hybrid algorithms may be generated for a \( 2^p \times 2^p \) mesh, each of which performs \( p-K \) phases of CE, for some \( K \) in the range \( 0 \ldots p \), and then a single DE operation in each \( 2^K \times 2^K \) submesh. Thus, \( K = 0 \) corresponds to pure CE while \( K = p \) corresponds to pure DE. Since phases involving larger meshes require more startups, use of combining in these phases helps in minimizing the overall number of message startups. When the importance of minimizing startup costs decline, due to the decrease in submesh size, the change to DE helps in decreasing transmission volume. Thus, this scheme of hybridizing exploits the benefits of both the combining and the direct exchange approaches.

![Decision Tree](image)

Figure 2.8: The decision tree for switching to Direct Exchange in a hybrid scheme.

The choice of a suitable \( K \) that minimizes the completion time among the various possible \( K \)-values in the range \( 0 \ldots p \) depends on system factors such as mesh size, startup cost, etc. and also on the problem size \( m \). It may appear that the decision to change to DE could be made on a phase-by-phase basis. That is, at the start of a given phase \( j \), CE or DE may be chosen for that phase by comparing the completion
times of CE and DE for that submesh size and effective block size. This is equivalent to comparing just two paths in the decision tree, one being DIRECT, the other being pure CYCLIC (see Figure 2.8). This ignores other paths in the tree of the form (CYCLIC...CYCLIC.DIRECT), which may be faster for the given submesh size and effective block size. Thus, a local decision is not possible. All the different paths in the tree need to be compared before deciding the best path.

One way to do this analysis is to traverse the tree bottom-up. Since the submesh size and effective block size can be computed for phase 0, the decision whether to perform that phase with CE or DE can be made. If CE is decided upon, all the $p$ phases need to be CE. Otherwise, the tree node for phase 1 is examined and a decision needs to be made then, and so on. The key observation is that, if a phase $\pi$ is best performed using CE, then the previous phases $\pi + 1, \pi + 2, \ldots, p - 1$ also need to be done using CE. But, if $\pi$ is best performed with DE, that does not imply anything about the decisions 'up' the tree.

An equivalent approach to analyze all the paths of the tree is to develop a quantitative model of the hybrid scheme, with $K$ as a dependent parameter. This would allow the computation of the completion times of all possible hybrids and choose the best one. The model is derived below under the assumption that both CE and DE are performed with barriers between steps. As described in section 2.3.2, we have also developed models for CE that either do not impose barriers at all or impose them only between phases (not steps).

The completion time of the hybrid algorithm can be analytically modeled in terms of the primary factors - startup and transmission time - and the secondary factors - reorder and barrier time. The CE component of the hybrid algorithm has $p - K$
phases, each with two message startups, so that the time spent in message startups in the CE phase is $S^C_E = 2(p - K)t_s$. The time taken by message transmissions is

$$T^{CE} = \begin{cases} (2^p - 2^K)2^{2p-1}mt_p & \text{if } K > 0 \\ 2^{3p-1}mt_p & \text{else} \end{cases}$$

The CE component also has a data reordering overhead of $R^{CE} = \max((p - K - 1)2^{p-1}mt_{mem}, 0)$ and has a penalty of $B^{CE} = (p - K - 1)b$ in barrier synchronization.

The DE component of the hybrid algorithm operates on a $2^{K-1} \times 2^{K+1}$ mesh and hence has $2^{4K-2}$ steps. So, the time spent by it in message startups is $S^{DE} = 2^{2K}t_s$. Message transmissions in the DE component take a time of $2^{3K-2}2^{2p-2K}mt_p$ if $K > 0$, since the size of each message in the DE component is $2^{2p-2K}m$. In general, the message transmission time can be expressed as

$$T^{DE} = \begin{cases} 2^{2p+K-2}mt_p & \text{if } K > 0 \\ 0 & \text{else} \end{cases}$$

Between the CE component, if any, and the DE component, the data array needs to be reordered so as to extract the first message. This overhead amounts to

$$R^{DE} = \begin{cases} 2^{2p-2K}mt_{mem} & \text{if } K < p \\ 0 & \text{else} \end{cases}$$

Once this is done, the data for subsequent messages can be extracted in parallel with the previous communications while the data for the executing processor itself can be extracted in parallel with the last communication. If $t_{mem} \leq t_p$, these additional reordering times would be completely subsumed in the transmission time and hence would not contribute to $R^{DE}$. The barrier synchronizations between steps impose an additional cost of $B^{DE} = (2^{3K-2} - 1)b$. The time incurred by each of the components of the hybrid algorithm can then be expressed as the sum of these terms i.e. $T_{hybrud} =$
\[ S^{CE} + T^{CE} + R^{CE} + B^{CE} \text{ and } T^{DE}_{\text{hybrid}} = S^{DE} + T^{DE} + R^{DE} + B^{DE}, \text{ so that} \]

\[
T_{\text{hybrid}} = \begin{cases} 
2pt_s + 2^{3p-1}mt_p + (p-1)2^{2p-1}mt_{\text{mem}} + (p-1)b & \text{if } K = 0 \\
2^{3p-2}t_s + 2^{3p-2}mt_p + (2^{3p-2} - 1)b & \text{if } K = p \\
\{2(p - K) + 2^{3K-2}\}t_s + (2^{3p-1} - 2^{2p+K-2})mt_p + \\
\{(p - K - 1)2^{2p-1} + 2^{2p-2K}\}mt_{\text{mem}} + (p - K - 1 + 2^{3K-2})b & \text{else} 
\end{cases}
\]

(2.8)

This expression is a function of \( K \) and the value \( K_{\text{opt}} \) for which it attains the minimum can be computed by evaluating this expression for each of the \( p + 1 \) possible values of \( K \), that is 0,...,p. Thus, the best algorithm in the family of hybrid algorithms can be chosen analytically.

The analytical model can be used to predict the variation of \( K_{\text{opt}} \) when the underlying system parameters and the problem size vary. The following theorems are based on the analytical model and offer insight into the trends exhibited by \( K_{\text{opt}} \) when a single parameter is changed, with all others held constant.

**Theorem 1:** Increasing \( m \) cannot decrease \( K_{\text{opt}} \), all other factors being equal (that is, the DE component of the hybrid will increase or stay the same when data block size is increased).

**Proof:** Consider the finish time of the hybrid for a given \( m \) and \( K > 0 \), denoted as \( T_{\text{hybrid}}(m, K) \). Since \( K_{\text{opt}} \) is the value of \( K \) which results in the least completion time, it follows that

\[
\forall K \in \{0, \ldots, p\} : T_{\text{hybrid}}(m, K_{\text{opt}}) \leq T_{\text{hybrid}}(m, K) \tag{2.9}
\]

It will be shown that

\[
\forall m' > m : \forall K < K_{\text{opt}} : T_{\text{hybrid}}(m', K_{\text{opt}}) < T_{\text{hybrid}}(m', K) \tag{2.10}
\]
In other words, choosing \( K < K_{opt} \) for \( m' > m \) cannot be an optimal choice as it results in a larger completion time than just retaining the old optimal value \( K_{opt} \).

The theorem follows from this.

The inequality 2.10 is proven by considering three distinct cases. For \( K_{opt} = 0 \), there is no \( K < K_{opt} \), so that it is vacuously true. For \( 0 \leq K < K_{opt} < p \), we have

\[
T_{hybrid}(m', K_{opt}) = \{2(p - K_{opt}) + 2^3K_{opt}^{-2}\}t_s + (2^3p^{-1} - 2^{2p+K_{opt}^{-2}})m't_p + \{p - K_{opt} - 1\}2^{2p-1} + 2^{2p-2K_{opt}}\}
\]

\[
T_{hybrid}(m'. K_{opt}) = T_{hybrid}(m, K_{opt}) + (2^3p^{-1} - 2^{2p+K_{opt}^{-2}})(m' - m)t_p +
\]

\[
\{(p - K_{opt} - 1)2^{2p-1} + 2^{2p-2K_{opt}}\}(m' - m)t_{mem} \tag{2.11}
\]

A similar expression can be derived for \( T_{hybrid}(m'. K) \) as

\[
T_{hybrid}(m', K) = \{2(p - K) + 2^3K^{-2}\}t_s + (2^3p^{-1} - 2^{2p+K^{-2}})m't_p
\]

\[
+\{(p - K - 1)2^{2p-1} + 2^{2p-2K}\}m't_{mem} + (p - K - 1 + 2^{3K^{-2}})b
\]

which can be rewritten as

\[
T_{hybrid}(m', K) = T_{hybrid}(m, K) + (2^3p^{-1} - 2^{2p+K^{-2}})(m' - m)t_p +
\]

\[
\{(p - K - 1)2^{2p-1} + 2^{2p-2K}\}(m' - m)t_{mem} \tag{2.12}
\]

Since \( K < K_{opt} \), the coefficient of \( (m' - m)t_p \) for \( T_{hybrid}(m'. K_{opt}) \) in equation 2.11 is smaller than that for \( T_{hybrid}(m', K) \) in equation 2.12. Similarly, the coefficient of \( (m' - m)t_{mem} \) is also smaller with \( T_{hybrid}(m', K_{opt}) \) than with \( T_{hybrid}(m', K) \). From these observations and from inequality 2.9, it follows that \( T_{hybrid}(m', K') > T_{hybrid}(m', K_{opt}) \).
Finally, for the case where $0 < K < K_{opt} = p$, we have

\[ T_{hybrid}(m', K_{opt}) = T_{hybrid}(m, K_{opt}) + 2^{3p-2}(m' - m)t_p \]

Here again, the coefficient of $(m' - m)t_p$ is smaller than that for $T_{hybrid}(m', K)$. While there is no term containing $t_{mem}$ unlike equation 2.12. Inequality 2.10 follows from this. Q.E.D. #

**Theorem 2:** Increasing $t_s$ cannot increase $K_{opt}$, all other factors being equal (that is, the DE component of the hybrid will decrease or stay the same when the message startup time increases).

**Proof:** Let $T_{hybrid}(t_s, K')$ be the completion time of a hybrid algorithm where all parameters except $t_s$ and $K$ are held constant. From the definition of $K_{opt}$, it follows that

\[ \forall K \in \{0, \ldots, p\} : T_{hybrid}(t_s, K_{opt}) \leq T_{hybrid}(t_s, K) \tag{2.13} \]

It will be shown that

\[ \forall t'_s > t_s : \forall K > K_{opt} : T_{hybrid}(t'_s, K_{opt}) < T_{hybrid}(t'_s, K) \tag{2.14} \]

From this, the theorem follows by the same reasoning as in Theorem 1.

The inequality 2.14 is proven by considering three distinct cases. For $K_{opt} = p$, there is no $K > K_{opt}$ so that it is vacuously true. For $0 < K_{opt} < K < p$, we have

\[ T_{hybrid}(t'_s, K_{opt}) = T_{hybrid}(t_s, K_{opt}) + \{2(p - K_{opt}) + 2^{2K_{opt} - 2}\}(t'_s - t_s) \tag{2.15} \]

and

\[ T_{hybrid}(t'_s, K) = T_{hybrid}(t_s, K) + \{2(p - K) + 2^{3K - 2}\}(t'_s - t_s) \tag{2.16} \]
Consider the difference of the coefficients of the $t'_s - t_s$ term in the two expressions above: $(2^{3Kopt} - 2^{3Kopt - 2}) + 2(Kopt - K)$. For $K > Kopt \geq 1$, this difference is positive. So, the coefficient in expression 2.16 is larger than that in expression 2.15. Together with inequality 2.13, this implies that $T_{\text{hybrid}}(t'_s, K) > T_{\text{hybrid}}(t'_s, Kopt)$.

Finally, the case $Kopt = 0$ is considered.

$$T_{\text{hybrid}}(t'_s, Kopt) = T_{\text{hybrid}}(t_s, Kopt) + 2p(t'_s - t_s)$$

Here too, the coefficient of the $t'_s - t_s$ term is smaller than that in expression 2.16. Hence $T_{\text{hybrid}}(t'_s, K) > T_{\text{hybrid}}(t'_s, Kopt)$. Q.E.D. #

**Theorem 3:** For sufficiently large $m$, increasing $p$ cannot decrease $Kopt$, all parameters except $p$ being equal (that is, the DE component of the hybrid will increase or stay the same when the mesh size is increased, provided the block size is sufficiently large).

**Proof:** Let $T_{\text{hybrid}}(p, K)$ be the completion time of a hybrid algorithm where all parameters except $p$ and $K$ are held constant. It needs to be shown that

$$\exists m_c \text{ s.t. } \forall m > m_c : \forall p' > p : \forall K < Kopt : T_{\text{hybrid}}(p', Kopt) < T_{\text{hybrid}}(p, K) \quad (2.17)$$

By the same reasoning as in Theorem 1, this would imply the theorem.

The inequality above is proven by considering three distinct cases. For $Kopt = 0$, there is no $K < Kopt$, so that the inequality is vacuously true. For the case $0 < K < Kopt \leq p$, we have

$$T_{\text{hybrid}}(p', K) - T_{\text{hybrid}}(p, Kopt) = \left\{(2^{3K-2} - 2^{3Kopt-2}) + 2(Kopt - K)\right\}t_s + \left\{(2^{2p'-2K} - 2^{2p'-2Kopt}) + 2^{2p'-1}(Kopt - K)\right\}mt_{mem} + \left\{(2^{3K-2} - 2^{3Kopt-2}) + (Kopt - K)\right\}b$$

29
Since $K < K_{opt}$, the coefficients of $mt_p$ and $mt_{mem}$ are strictly positive. Also, in
the coefficients of $t_s$ and $b$, the terms involving $K_{opt} - K$ are positive. The only
negative component in the above expression is $(2^{3K^{3-2}} - 2^{3K_{opt}^{2-2}})(t_s + b)$. Note that
this component does not involve $m$ while two of the positive terms do involve it.
Thus, for a sufficiently large $m$, the above difference will be strictly positive i.e.
$T_{hybrid}(p', K_{opt}) < T_{hybrid}(p', K)$.

Finally, in the case $0 = K < K_{opt} < p$, the difference would be

$$T_{hybrid}(p', 0) - T_{hybrid}(p', K_{opt}) = (2K_{opt} - 2^{3K_{opt}^{3-2}})t_s + 2^{2p' + K_{opt}^{2-2}}mt_p +$$

$$(K_{opt} - 2^{3K_{opt}^{3-2}})b + (2^{2p' - 1}K_{opt} - 2^{2p' - 2}K_{opt})mt_{mem}$$

The terms involving $m$ are strictly positive and those not involving it are non­
positive. So, for sufficiently large $m$, the difference would be strictly positive i.e.
$T_{hybrid}(p', K_{opt}) < T_{hybrid}(p', K)$. Q.E.D.

**Theorem 4:** Increasing $b$ cannot increase $K_{opt}$, all other factors being equal (that is.
the DE component of the hybrid will decrease or stay the same when the cost of a
barrier synchronization is increased).

**Proof:** Let $T_{hybrid}(b, K)$ be the completion time of a hybrid algorithm where all pa­
rameters except $b$ and $K$ are held constant. From the definition of $K_{opt}$, it follows
that

$$\forall K \in \{0, \ldots, p\} : T_{hybrid}(b, K_{opt}) \leq T_{hybrid}(b, K)$$

(2.18)

It will be shown that

$$\forall b' > b : \forall K > K_{opt} : T_{hybrid}(b', K_{opt}) < T_{hybrid}(b', K)$$

(2.19)

From this, the theorem follows by the same reasoning as in Theorem 1.
The inequality 2.19 is proven by considering three distinct cases. For $K_{opt} = p$, there is no $K > K_{opt}$, so that it is vacuously true. For $0 < K_{opt} < K < p$, we have

$$T_{hybrid}(b', K_{opt}) = T_{hybrid}(b, K_{opt}) + (p - K_{opt} - 1 + 2^{3K_{opt}-2})(b' - b)$$

(2.20)

and

$$T_{hybrid}(b', K) = T_{hybrid}(b, K) + (p - K - 1 + 2^{3K-2})(b' - b)$$

(2.21)

For $K > K_{opt} > 0$, $2^{3K-2} - K > 2^{3K_{opt}-2} - K_{opt}$. So, the coefficient of the $b' - b$ term is bigger in expression 2.21 than in expression 2.20. This observation, together with inequality 2.18 show that $T_{hybrid}(b', K_{opt}) < T_{hybrid}(b', K)$.

Finally, considering the case $K_{opt} = 0$, we have

$$T_{hybrid}(b', K_{opt}) = T_{hybrid}(b, K_{opt}) + (p - 1)(b' - b)$$

where the coefficient of the $b' - b$ term is again smaller than with expression 2.21. Hence, $T_{hybrid}(b', K_{opt}) < T_{hybrid}(b', K)$. Q.E.D.

<table>
<thead>
<tr>
<th>Factor being varied</th>
<th>CE Finish Time</th>
<th>DE Finish Time</th>
<th>$K_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \uparrow$</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>$t_s/t_p \uparrow$</td>
<td>↑</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>$p \uparrow$</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>$b \uparrow$</td>
<td>↑</td>
<td>↑</td>
<td>↓</td>
</tr>
</tbody>
</table>

Table 2.2: Variation in completion times of CE and DE, and also in $K_{opt}$, as the parameters of complete exchange are varied one at a time. Among CE and DE, the one whose finish times increases the most is indicated with ↑.

These trends are summarized in Table 2.2.

In the following section, these trends predicted by the model are validated with a simulation study.
2.5 Simulation results and discussion

The family of hybrid algorithms needs to be evaluated to find the least completion time among the hybrids. The evaluation, in general, may be carried out using analysis, simulation or measurement of an implementation on a multicomputer. All the three avenues were pursued as they have their own merits.

Analytical models were developed for Direct Exchange (DE) and Cyclic Exchange (CE). The model for DE assumes barriers between steps (Section 2.2.2). Three models were developed for CE: one that assumes no barriers, one assuming a barrier between steps, and one with barrier between phases. The former two were implemented and studied, whereas the latter was used primarily for its simplicity of analysis. These were, in turn, used to develop a quantitative model of the hybrid algorithms. This model can be used to predict the best hybrid in the family for specific combinations of system factors and the problem size.

The predictions of the analytical model can be verified either with a simulation or with an implementation on a multicomputer. Simulation has the advantage that the architectural parameters can be systematically varied, which would not be possible in a real machine. An implementation, however, would provide concrete proof that the scheme is practically useful. So, both these methods were employed. The results of the simulation study are presented in this section and those of the implementation in Section 2.8.

The performance of CE, DE and their hybrids, with and without the use of barriers, were studied using simulation. The simulator was written in C. using the CSIM simulation package [35]. It models a wormhole-routed network with $k$-ary $n$-cube.
topology, at the flit level in detail. The simulator reports the completion time, the average latency of all messages and the average ideal (contention-free) latency.

Several parameters were varied for the simulation study using a combinatorial design. The system parameters are \( p \), a measure of the size of the \( 2^p \times 2^p \) mesh; \( t_s \), the message startup cost; and \( b \), the barrier cost. The problem size is the number \( m \) of bytes in each data block. The hybrids were investigated under three cases:

- without barriers.
- with barriers that utilize the hardware support of the underlying architecture to reduce barrier cost, if such support is available
- with barriers implemented in software, should the underlying machine not have support for barrier synchronization.

The following values were held constant: \( t_p \) at 0.012\( \mu \)s and \( t_{mem} \) at 0.007\( \mu \)s.

The objective of the study was to compare hybrid algorithms with different \( K \) values and determine:

- if the hybrid algorithms perform better than the pure algorithms for some parameter values.
- the best value of \( K \) for a given set of system parameters and problem size.
- the accuracy of the analytical models.

The following tables and plots summarize the results of the study.
Table 2.3: The actual (top row) and predicted (bottom row) completion times for various hybrids with hardware barriers, with barrier cost of 1μs. The numbers in bold correspond to the best K for that block size.
2.5.1 Results with hardware barrier

For $16 \times 16$ and larger meshes, the high degree of contention degrades performance.

This contention can be controlled by imposing a barrier between successive steps of CE or DE. This is found to improve performance, as demonstrated below. This solution is attractive because many modern multicomputers, such as the Cray T3D, offer hardware support for barriers, so that the cost of a barrier is low. This subsection focuses on the performance of hybrid algorithms using such fast hardware barriers.

The cost of a barrier is held fixed at $b = 1 \mu s$.

Table 2.3 presents the data for hybrids with barriers between successive steps for two mesh sizes and two startup ratios. The startup ratio of 7000 is representative of systems available widely today, while the ratio of 150 is a low value that may be expected in future systems. The top row for each $K$ gives the actual completion time of the simulation while the bottom row provides the completion times predicted by the analytical model. Note that $K = 0$ corresponds to CE while $K = p - 1$ corresponds to DE.

It can be seen from Table 2.3(a) for a $16 \times 16$ mesh that there is very close correspondence between analytically predicted and observed results. The model predicts that, for a fixed mesh size and startup time, a hybrid algorithm should take longer to finish as $m$ increases, due to the increase in transmission time component. This is indeed the case. Theorem 1 derived from the model also predicts that, as $m$ increases, the best hybrid would tend to have an increased DE component i.e. the best $K$, denoted $K_{opt}$, would tend to increase. This is also confirmed. Table 2.3(b) confirms these trends for a larger system size of $32 \times 32$. The model has been verified.
to be accurate for a large range of mesh size, block size and startup ratios, with and without barriers.

Table 2.4 illustrates the variation of $K_{opt}$ with $m$, $t_s$ and $p$. Increasing $m$ while holding the other two constant leads to an increase in $K_{opt}$ as already noted. If $t_s$ is increased, all other factors being equal, $K_{opt}$ tends to decrease, which is in accordance with Theorem 2. Finally, if $p$ alone were increased, $K_{opt}$ tends to increase, favoring DE as stated in Theorem 3. This is in agreement with the predictions of the model.

<table>
<thead>
<tr>
<th>mesh size</th>
<th>$t_s/t_p$</th>
<th>8</th>
<th>32</th>
<th>128</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 8</td>
<td>150</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16 x 16</td>
<td>150</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>32 x 32</td>
<td>150</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.4: The $K$ of the best hybrid with hardware barriers, with barrier cost of 1 $\mu s$.  

2.5.2 Results with software barriers

The results and analysis presented so far assume a constant barrier cost of 1 $\mu s$. If the target machine does not provide hardware support for barriers, or if the processors involved in the complete exchange are not contiguous, the barrier would have to be implemented in software. Barrier synchronization algorithms are usually tree-structured, with a reporting stage and a wakeup stage. On a $2^p \times 2^p$ mesh, each
of these stages take a time of \(2pt_s\), for a total of \(4pt_s\). This higher cost of barriers will affect the completion times of the algorithms. However, the impact on CE will be much lower than on DE, as the former employs far fewer barriers. This, in turn, implies that the hybrid with lower DE component will be favored when the barrier cost is high i.e. \(K_{opt}\) would tend to decrease (as proved in Theorem 4). This subsection presents and analyzes the performance of hybrid algorithms with software barriers.

For Direct Exchange (DE) on a \(2^p \times 2^p\) \((p > 1)\) mesh, there are \(2^{3p-2}\) steps so that the total cost of barriers is \((2^{3p-2} - 1)4pt_s\). For Cyclic Exchange (CE), the recursive structure implies that the parameter \(p\) decreases in successive phases. The total barrier cost is \(2\{4pt_s + 4(p-1)t_s + \ldots + 4.2t_s\}\) since there is no need for barriers in the last \(2 \times 2\) phase. The factor of 2 arises because there are two barriers in each phase. This sum evaluates to \(\{p(p+1) - 2\}4t_s\). The barrier cost for hybrids for \(0 < K < p\) can be calculated along similar lines. (Note that Theorem 4 would still hold true with this modified cost model for barrier synchronization since the number of barriers in DE is much higher than with CE.)

The completion times of the hybrids with software barriers were determined and the value of \(K_{opt}\) found, for each combination of \(p, m\) and the \(t_s/t_p\) ratio. The results are presented in Table 2.5 and exhibit similar variation with \(p, m\) and \(t_s/t_p\) as the hardware barrier case (Table 2.4), thus confirming the analytically proven theorems.

A software barrier is more expensive than a hardware barrier. Since DE uses more barriers than CE, it would be expected that the DE component of the best hybrid i.e. \(K_{opt}\) would decline when software barriers replace hardware barriers. Also, this decrease should be more marked at large values of \(m\) and \(p\), as these conditions lead to
a high $K_{opt}$. Furthermore, as software barrier cost rises with $t_s$, $K_{opt}$ may be expected to decline at high $t_s$ values also.

A comparison of Table 2.5(a) with Table 2.4(a) shows that $K_{opt}$ values tend to decrease when software barriers replace hardware barriers, especially for large $t_s$. Comparing all three subtables of Table 2.5 and Table 2.4 reveals that the decrease of $K_{opt}$ with software barriers is more pronounced for larger meshes. Thus, the predictions are borne out by the simulation study.

<table>
<thead>
<tr>
<th>mesh size</th>
<th>$t_s/t_p$</th>
<th>Data Block Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>$8 \times 8$</td>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
</tr>
<tr>
<td>$32 \times 32$</td>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.5: The $K$ of the best hybrid with software barriers.

To summarize, hybrid algorithms do quite well when barriers are used to control contention and, in fact, are the optimal algorithms for a large range of parameters. Also, the analytical model is useful in choosing the best $K$ for a given set of parameters.
2.5.3 Results without barrier

The performance of the hybrids in the absence of barriers in both the CE and DE components is reported in this section.

Table 2.6 presents the \( K \) for the best hybrid without barriers for three different mesh sizes and startup ratios. Though all the combinations of factors were studied, we present the data for only some selected combinations. The optimal algorithm for a large range of parameters is a hybrid algorithm, thus establishing that hybridizing is quite useful in the absence of barriers also.

| mesh | \( t_s/t_p \) | \begin{tabular}{c|llllll} Data Block Size (in bytes) \hline mesh & \( 8 \) & \( 32 \) & \( 128 \) & \( 1024 \) & \( 4096 \) & \( 16384 \) \hline \( 8 \times 8 \) & 150 & 1 & 2 & 3 & 3 & 3 & 3 \hline & 1000 & 0 & 0 & 1 & 3 & 3 & 3 \hline & 7000 & 0 & 0 & 0 & 2 & 3 & 3 \hline \( 16 \times 16 \) & 150 & 2 & 3 & 3 & 3 & 3 & 3 \hline & 1000 & 0 & 2 & 3 & 3 & 3 & 3 \hline & 7000 & 0 & 0 & 1 & 3 & 3 & 3 \hline \( 32 \times 32 \) & 150 & 3 & 3 & 3 & 3 & 3 & 3 \hline & 1000 & 2 & 3 & 3 & 3 & 3 & 3 \hline & 7000 & 0 & 1 & 3 & 3 & 3 & 3 \hline \end{tabular} |

Table 2.6: The \( K \)-value of the best hybrid without barriers.

Note that, although system size, block size and startup cost are varied by several orders of magnitude, the best hybrid tends to be \( K = 3 \). This is contrary to the trend in the case of hybrids with barriers, where \( K_{opt} \) tends to increase with \( m \) and \( p \). thus favoring DE. The reason for this surprising pattern is discussed in the next section.
Table 2.7: Comparison of completion times with and without hardware barriers for Direct Exchange and Cyclic Exchange on a 32 × 32 mesh.

### 2.6 Impact of contention on finish time

Wormhole routing reserves mesh links. When a message is forced to wait for a link to become available, the links which it holds are not released, causing other messages which need those links to block. This may have the effect of increasing message latency enormously and degrading performance. This section quantifies the degree of contention and explores its impact on the finish time of complete exchange.

The degree of contention in the execution of an algorithm is measured by the ratio of average actual latency of all messages in the algorithm to the average ideal (contention-free) latency of all messages. This ratio, called the *Contention Factor*, is meaningful only when all the messages in the algorithm are of the same length. This is the case for both the pure algorithms Cyclic Exchange (CE) and Direct Exchange.
(DE), though not for the hybrids. Thus, Figure 2.9 plots the contention factors only for the pure algorithms.

Figure 2.9 clearly shows the steep increase in contention in DE as $m$ increases, for a $32 \times 32$ mesh. This happens for a range of startup costs, from $1 \mu s$ to $50 \mu s$. In contrast, the contention factor for CE remains remarkably stable even when $m$ varies by 4 orders of magnitude. This accounts for the observation in Section 2.5.3 that, in the absence of barriers, the best hybrid algorithm always has 3 or fewer phases replaced by Direct Exchange (DE). This is due to the fact that, as $m$ increases, contention in DE increases steeply, thus offsetting any savings due to lower transmission volume. Thus, the best hybrid does not have a large DE component, even for large $m$.

![Figure 2.9](image)

Figure 2.9: Variation of contention factor with block size for Cyclic Exchange and Direct Exchange in the absence of barriers for a $32 \times 32$ mesh. Contention factor is defined as the ratio of average actual latency of all messages in the algorithm to the average ideal (contention-free) latency.

Another striking feature of the plot in Figure 2.9 is that the contention factor for DE is lower when startup time is higher i.e. higher message startup cost has
the effect of decreasing network contention. This is because increasing the startup time increases the interval between injection of messages into the network by each processor. This has the effect of decreasing the offered load (number of messages per unit time) on the network, thus leading to lower latency.

This leads to the counter-intuitive observation that increasing the startup cost can decrease the completion time by reducing contention. This is illustrated in Table 2.7a. which presents the completion times for DE with and without barriers for several values of $m$ and $t_s/t_p$. In the absence of barriers, for $m = 1024$ or 4096, increasing the startup ratio from 150 to 7000 decreases the completion time for DE, although it has a large number of startups. This seemingly strange behavior can be explained by noting the strong influence of contention on message latencies in DE. This is seen to happen with CE also, but only for $m = 32$ (Table 2.7b).

The high level of contention in the no-barrier version also explains why it is beneficial to impose barriers. The decrease in finish time due to reduction or the elimination in contention often offsets the increase due to the cost of the barriers. This is illustrated in Table 2.7. The performance of DE improves when hardware barriers are imposed for $m \geq 128$ at $r = 150$ and for $m \geq 4096$ at $r = 7000$. For low $m$, contention is not high: so, the improvement in performance by eliminating contention is outweighed by the cost of barriers. At higher values of $m$, the high degree of contention makes barriers useful. Also, barriers are more beneficial at lower startup costs, because contention is higher in that regime. For CE, the barrier version performs better than the non-barrier version for a large range of $m$ and for widely differing startup costs. This is consistent with the fact that the contention factor for CE hardly varies with $m$ or $t_s/t_p$.
2.7 Influence of barriers on $K_{opt}$

Table 2.7 validates the claim that using barriers can potentially improve performance. The degree to which they enhance performance depends on the cost of the barriers, which in turn is influenced by the availability of hardware support for barrier synchronization. This subsection investigates the effectiveness of barrier usage in the presence and absence of hardware support for barrier synchronization and also the impact on the $K$-value of the best hybrid. As noted earlier, the data for $8 \times 8$ mesh assume that barriers are used only between phases, while they are employed between successive steps (not phases) for larger meshes.

2.7.1 Impact of hardware barriers

Table 2.8 presents data on the best hybrid to use and whether or not it uses hardware barriers, for several combinations of $p$, $m$ and $t_s/t_p$. The barrier cost is fixed at $1\mu s$.

Table 2.8(a) shows that using hardware barriers is advantageous at larger values of $m$ for $8 \times 8$ meshes. Furthermore, the value of $m$ at which it becomes better to use barriers tends to increase with $t_s/t_p$. This is explained by noting that large $t_s$ values serve to reduce contention, as discussed in Section 2.6, so that barriers are not as useful.

From Table 2.8(b), it can be seen that it is better to employ barriers for $16 \times 16$ meshes except for $t_s/t_p = 7000$, $m \leq 32$. This exception arises because, at large $t_s$ and small $m$, the degree of contention is too small to justify the use of barriers. However, the barrier version takes only 3.7% extra time for $m = 8$ and 7.1% more for $m = 32$. 
Table 2.8: The best $K$ and hardware barrier usage for various $p$, $m$ and $t_s/t_p$. The letter “h” denotes the use of hardware barriers while “n” indicates absence of barriers.

Finally, the data of Table 2.8(c) for $32 \times 32$ meshes show that it is uniformly better to use hardware barriers.

Thus, use of hardware barriers leads to better performance in most cases: for relatively small mesh sizes, high $t_s$ and small $m$, it increases the finish time by only a small percentage. So, if the target system provides hardware support for barrier synchronization, it is a good strategy to use it in all cases.

### 2.7.2 Impact of software barriers

When barriers need to be implemented in software because of lack of architectural support for barrier synchronization, the decision on their usage is not as clearcut as with hardware barriers. This is illustrated further in the following subsection.

Table 2.9 presents the choice of best $K^*$ and whether or not it uses software barriers, for several combinations of $p$, $m$ and $t_s/t_p$. Part (a) shows that software barriers are useful only when $t_s$ is low and $m$ is high for $8 \times 8$ meshes. At higher values of $t_s$, the
<table>
<thead>
<tr>
<th>mesh size</th>
<th>$t_s/t_p$</th>
<th>8</th>
<th>32</th>
<th>128</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 8</td>
<td>150</td>
<td>n</td>
<td>2 n</td>
<td>3 n</td>
<td>3 n</td>
<td>3 s</td>
<td>3 s</td>
</tr>
<tr>
<td>1000</td>
<td>0 n</td>
<td>n</td>
<td>n</td>
<td>1 n</td>
<td>3 n</td>
<td>3 n</td>
<td>3 n</td>
</tr>
<tr>
<td>7000</td>
<td>0 n</td>
<td>n</td>
<td>n</td>
<td>0 n</td>
<td>2 n</td>
<td>3 n</td>
<td>3 n</td>
</tr>
<tr>
<td>16 x 16</td>
<td>150</td>
<td>2 n</td>
<td>3 n</td>
<td>2 s</td>
<td>3 s</td>
<td>4 s</td>
<td>4 s</td>
</tr>
<tr>
<td>1000</td>
<td>0 n</td>
<td>2 n</td>
<td>3 n</td>
<td>2 s</td>
<td>2 s</td>
<td>3 s</td>
<td>3 s</td>
</tr>
<tr>
<td>7000</td>
<td>0 n</td>
<td>1 n</td>
<td>3 n</td>
<td>1 s</td>
<td>2 s</td>
<td>2 s</td>
<td>3 s</td>
</tr>
<tr>
<td>32 x 32</td>
<td>150</td>
<td>1 s</td>
<td>1 s</td>
<td>2 s</td>
<td>3 s</td>
<td>4 s</td>
<td>5 s</td>
</tr>
<tr>
<td>1000</td>
<td>2 n</td>
<td>1 s</td>
<td>1 s</td>
<td>2 s</td>
<td>3 s</td>
<td>4 s</td>
<td>4 s</td>
</tr>
<tr>
<td>7000</td>
<td>0 n</td>
<td>1 n</td>
<td>3 n</td>
<td>1 s</td>
<td>2 s</td>
<td>3 s</td>
<td>3 s</td>
</tr>
</tbody>
</table>

Table 2.9: The best $K$ and software barrier usage for various $p$, $m$ and $t_s/t_p$. The letter “s” denotes the use of software barriers while “n” indicates absence of barriers.

The cost of a software barrier is so high that it is better to dispense with it. Similarly, at low $m$, the level of contention is not high enough to accept the penalty of barriers.

It is observed in Table 2.9(b) for 16 x 16 meshes also that software barriers are beneficial at higher $m$. The crossover point at which barriers become useful is seen to shift towards higher $m$ as $t_s$ increases, because large $t_s$ tends to reduce contention, thus making barriers less useful.

The same trend is also seen in part (c) for 32 x 32 meshes. Contrasting parts (b) and (c) reveals that the crossover point tends to decrease as $p$ increases, as the level of contention is higher in larger meshes and barriers reduce it.

Summarizing, barriers can be effective even when they need to be implemented in software. Software barriers are most useful for larger meshes, lower startup costs and higher data block sizes. As the overall trend is towards systems with lower message startup time and increasing number of processors, and towards larger applications.
barriers may be expected to be useful for complete exchange even in the absence of architectural support for them.

2.8 Experimental results

The family of hybrid algorithms were implemented on the Cray T3D system at the Ohio Supercomputer Center. It is a distributed memory machine with the interconnect being a 3-dimensional torus. This installation of T3D has 128 processing nodes, structured as $8 \times 4 \times 4$. The message-passing model is supported on the T3D with a native implementation of the Parallel Virtual Machine (PVM) [19] communication library. So, the hybrids were implemented using PVM. Certain factors had to be taken care of in the implementation: these are described in Appendix B.

The $K_{opt}$ of the hybrids that had the best performance for a given mesh size and block size are presented in Table 2.10. For a $2 \times 2$ mesh, CE does best for block sizes $8 \leq m \leq 128$, but for larger $m$, DE fares better. Thus, a hybrid model that switches from one to the other as appropriate would have the best performance for the entire range of $m$. For $4 \times 4$ meshes, CE does best for $8 \leq m \leq 32$. For $128 \leq m \leq 256$, the best algorithm is neither CE nor DE, but a hybrid algorithm: for larger $m$, DE fares better. Similarly, for $8 \times 8$ meshes also, the hybrid model switches smoothly from CE to DE through various intermediate hybrids.

These observations are consistent with the trends seen in the simulation study and the theorems derived from the analytical model. CE tends to do well at lower block sizes since the savings in message startup costs offset the higher volume of data transmission. DE fares well at larger block sizes since the volume of data handled by the network becomes important in this regime. The DE component tends to increase
<table>
<thead>
<tr>
<th>logical mesh size</th>
<th>Data block size (in bytes)</th>
<th>8</th>
<th>32</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 x 4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8 x 8</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.10: The $K$ of the best performing hybrid on the Cray T3D. Note that $8 \times 8$ mesh is embedded in a $8 \times 4 \times 4$ torus.

when the mesh size increases and the block size is held constant, as observed in the simulation study and the analytical model.

Thus, the experimental results from the implementation of the hybrid scheme on the Cray T3D multicomputer confirms and validates the thesis that exploiting the complementary strengths of combining and direct algorithms leads to lower execution times across a broad range of mesh sizes and block sizes.

### 2.9 Related Work

Considerable work has been done to find good solutions for the problem of performing complete exchange efficiently, since it occurs in numerous practically relevant applications such as FFT and parallel sorting. A review of past literature and their differences in approach with this work is provided in this section.

The complete exchange problem has been extensively studied on various topologies such as hypercubes [24, 7, 8, 6, 1, 34, 42] and tori [40]. Solutions have also been formulated for linear arrays and rings [12] and other architectures [28, 32] with circuit-switching. Several studies have focussed on the 2D mesh architecture [10, 36, 44, 43, 41].
Another class of studies has aimed at an efficient solution to complete exchange on specific parallel computers [45, 7, 34]. Many papers have analyzed the problem in the context of specific applications such as parallel sorting [2, 1, 25, 39].

Efficient implementation of complete exchange is a significant issue for designers of collective communication libraries. The question of performing collective communication efficiently has been addressed in [3, 5]. A survey of collective communication algorithms, including those for complete exchange, is given in [29].

Multiphase algorithms have been proposed for the hypercube [8] and these same algorithms have been implemented on a 2D mesh [9].

This work has several features that distinguish it from past studies. It hybridizes two algorithms that have been specifically formulated for meshes: Cyclic Exchange and Direct Exchange. It also provides an analytical model, unlike earlier works, which helps in choosing the best solution out of a family of hybrid algorithms. The predictions of the analytical model have been verified by both simulation and implementation on a Cray T3D to be accurate. One of the unique features of the study is the accurate modeling of message contention on a 2D mesh when barrier synchronization is employed, in contrast to many previous studies where the time loss due to contention is quantified in an approximate manner. It also demonstrates that link contention is an important factor in determining the finish time of complete exchange. This leads to two other novel observations. The first is that the finish time can be reduced for many parameter ranges by controlling contention through deploying barrier synchronizations between steps. The other is the counter-intuitive observation that increased message startup overhead can reduce the completion time.
by decreasing message contention. These observations are quantified and studied with the analytical model and further confirmed with a simulation study.

2.10 Conclusions

This chapter has proposed a scheme to hybridize the best known combining algorithm (Cyclic Exchange) and the best direct algorithm (Direct Exchange) for complete exchange in a way that effectively exploits their complementary merits. Analytical models have been proposed that quantify the influence of various factors on the completion time of CE (without barriers, with barriers between phases and with barriers between steps), DE (with barriers between steps) and their hybrids. The best hybrid for specified sets of system parameters and problem size can be chosen using the analytical model (when barriers are used). The hybrids were shown to perform better than previously known efficient algorithms for complete exchange for a large range of system and problem parameters, through both a simulation study as well as implementation on a Cray T3D multicomputer.

This study also brings out the strong influence that link contention among messages has on the performance of complete exchange algorithms. In the case of CE, analytical models are developed that identify patterns of message contention and quantify them. These are used to motivate the use of barriers between algorithmic steps to control contention. The predictions of the analytical model are validated with a simulation study, which further confirm the impact of contention on performance. The direct exchange approach is found to suffer from a large degree of contention in the absence of barriers. This in turn motivates the use of barriers, with or without support for them at the hardware level, to control contention and improve
performance. The conditions at which use of hardware and software barriers becomes beneficial are characterized and explained. The counter-intuitive phenomenon of decreased completion time with increased startup cost is identified and explained.
CHAPTER 3

INCREMENTAL PARALLELIZATION OF STENCIL COMPUTATIONS

3.1 Introduction

The Message Passing Interface (MPI) \cite{15, 20, 37} has been established as a standard way for parallel programs to access the services of the underlying parallel computing system. This has made it possible for the same parallel program to be executed on diverse computing platforms, including shared address space multiprocessors and distributed memory multicomputers. So, there is a strong impetus to rewrite sequential programs in a parallel form using MPI as this ensures portability across all parallel computing systems.

However, MPI assumes a distributed address space model which is difficult to program in. Converting a sequential program into a such a parallel form involves two aspects. The first is an explicit partitioning of the data into the separate address space of each process. There may be a need to repartition the data through interprocessor communication as the program executes. The second aspect concerns the communication and synchronization among the processors. The converted program needs to have explicit calls to communication primitives, identifying the set of data
elements to be sent/received and the identifier of the communicating processor. Similarly, it also needs to have explicit synchronization calls identifying the processor(s) that are to be synchronized with.

Tools that automatically generate a parallel numerical program from its sequential counterpart are available but they tend to produce parallel programs that often have considerably poorer performance than manually converted code. However, manual parallelization of a sequential program is a tedious and error-prone process that frequently involves rewriting the program from scratch. Ensuring that the parallelized program is correct and that it produces the same output as the sequential program for the same input is a challenging problem. This is particularly so in the case of legacy codes if the person who is parallelizing the program is not familiar with the program.

To address the twin issues of minimizing conversion effort and ensuring complete accuracy of the parallel program, it is essential to follow a systematic approach that eases manual implementation by synthesizing the parallel program through a series of intermediate steps while also ensuring accuracy by validating the output at each step. As errors can be introduced during the conversion, the approach should incorporate structures and processes for tracking these errors.

This chapter presents a case study in which a large application in computational fluid dynamics was manually parallelized using an incremental subroutine-by-subroutine approach that ensured the complete accuracy of the output at each intermediate step. It then proposes an automatable methodology for parallelizing stencil computations by abstracting the basic approach of the case study. By focusing on a well-defined but broad class of applications, the methodology avoids the performance
drawbacks associated with general tools for automatic parallelization. The methodology applies to nonrecursive programs that perform stencil computation (that is, each subscript in a use of an array is either identical with the corresponding one in each of its reaching definitions or differs from it only by a constant integer) with loops of unit stride, in which each array subscript is either an integer or a loop variable (in the case of a reference that defines an array) or of the form \(i \pm s\) (for a reference that uses an array), where \(i\) is a loop variable and \(s\) is an integer. A large class of applications based on finite differences conforms to these requirements. It is a three-pass approach, in which the first pass, called the Preliminary Pass, addresses two aspects. One is to ensure that the given program conforms to the requirements outlined above. The second is to let the user choose a tiled distribution for each dimension of each array across a pool of virtual processors. It is ensured algorithmically that this distribution is in conformance with the "owner computes" rule. The second pass, called the Blocking Pass, builds an augmented sequential program that is amenable to parallelization. The main task here is the programmatic conversion of each sequential routine into a parallelizable counterpart that simulates the action of many virtual processors on the distributed arrays. Communication among the processors is simulated by explicit copying of appropriate array elements. The conversion is incremental, as it traverses the call tree of the program in a bottom-up manner, augmenting each original routine with its converted equivalent. The resulting program, called the blocked form, retains the original routines as well as the new ones that access the tiled arrays, in order to compare the output of the sequential and the blocked code. The third pass, called the MPI Pass, is also automatable and adds communication calls to build an SPMD program. This approach offers several benefits. Since the blocked form is a sequential
program, the well-tested methods for debugging sequential programs can be applied to it to identify possible discrepancies between the sequential and the blocked code. Furthermore, since the first pass deals with data distribution while the second pass deals with the details of communication, these two issues are isolated. The incremental nature of the conversion allows user intervention to tune performance at many levels of the program. Since the comparison between the original and the blocked versions of each routine is retained till the last step in the MPI Pass, errors that may be introduced in the intervention will get located early in the conversion. Finally, in the event of a discrepancy between the original and the converted code, methods are provided to systematically track and locate the region of the code that causes the discrepancy.

The chapter is organized as follows.

3.2 The Case Study

3.2.1 Background

The case study involves a stencil computation in computational fluid dynamics, called *fdl3di*, that simulates the flow of air around an aircraft. The application essentially solves the Navier-Stokes partial differential equations by discretizing the volume to be simulated into a three-dimensional grid and evaluating the physical variables at each grid point. Numerical evaluation of the differential equations requires the solution of a system of simultaneous linear equations, which is performed using the three-dimensional Alternating Directions Implicit (3D ADI) method. This requires a traversal along each of the three dimensions, which are labeled I, J and K respectively. Each such traversal, called a *sweep*, sets up a system of pentadiagonal linear equations
by coupling successive values along the sweep direction. For example, for each value of K and J, the sweep along I sets up a system of equations using successive values in the I direction.

The main program of *fdl3d* is structured as a two-way nested loop. Each iteration of the outer loop contains many subiterations of the inner loop and ends with the propagation of boundary values. Each iteration of the inner loop carries out all the three sweeps and then computes the L2 norm of one of the arrays as the residual. In each run of the program, a set of 9 arrays describing the positions, velocities and other physical attributes of the particles are read as input, and a fixed number of iterations of the outer and inner loop are carried out, followed by an output of the 9 arrays. This allows the output of a run to be fed as input for the next run. In addition, at the end of each iteration and subiteration, the residuals are written out, though they do not play a role in further computation.

The program consists of 50 routines exceeding 5100 lines of FORTRAN code that manipulate 41 arrays whose dimensionality varies from one to five. Denoting the size of the grid in the three dimensions as I MAX, J MAX and K MAX respectively, the space complexity of the program is in $O(I \text{ MAX} \times J \text{ MAX} \times K \text{ MAX})$, so that large problem sizes do not fit in the memory of a uniprocessor system. Together with the need to reduce finish time, this provided an impetus to parallelize the program.

### 3.2.2 Need for Manual Parallelization

The principal issue involved in constructing a program based on a distributed address space model from its sequential counterpart is the partitioning of the global index sets of the original arrays into the local index sets of the arrays in the separate
address spaces of the processes constituting the SPMD program. For stencil computations, a good partitioning scheme is to tile the data space of each array into a number of blocks and map these blocks to the set of virtual processors. This subsection looks at the issues that arise in the choice of a tiled partitioning and shows that automated conversion using HPF will not fully exploit parallelism so that manual conversion is preferable.

A key component of \textit{fdl3di} is a subroutine called \textit{spenta} which is invoked in each of the sweeps for solving the pentadiagonal system of equations. For each sweep, \textit{spenta} is invoked within a loop which steps across a non-sweep dimension, called the principal dimension. The I sweep and the J sweep both have K as the principal dimension while the K sweep has J as the principal dimension. The subroutine is organized in two parts: a forward pass, in which the sweep dimension is traversed in ascending order of coordinates, and a backward pass, in which it is traversed in descending order. Each pass is a loop nest of depth two, with one of the loops stepping across the sweep dimension and the other along the non-principal dimension. The forward pass computes certain the elements of two arrays, called \textit{temp0} and \textit{wrk0}, which are used or redefined in the backward pass.

Two of the three loops involved in \textit{spenta} exhibit data dependencies that inhibit concurrency. Firstly, the loop stepping across the sweep dimension is inherently sequential as the values it produces in each iteration are consumed in the next iteration. Secondly, the arrays used in the subroutine do not have a dimension corresponding to the principal dimension so that there are output dependences between successive iterations of the principal loop. These dependences are not inherent in the structure.
of the computation but arise from the reuse of the same data structures in different iterations of a loop. Unless these output dependences are removed by a suitable expansion of the concerned arrays, the principal loop needs be executed serially.

These factors constrain the choice of the data distribution. Consider a distribution that blocks the arrays along a chosen dimension, say I. When executing spenta for I, all but one process in the parallel program must wait for its predecessor to finish its computations along the I loop and communicate the computed values to itself. Pipelined parallelism at the outermost loop, namely the principal loop K, is prohibited by the output dependences described above. It is possible to exploit pipeline parallelism in the loop that steps across the non-principal dimension but this would require frequent communication as this loop is not the outermost loop in its loop nest. Similar constraints arise when any other dimension or a combination of these are partitioned using a blocked distribution. Thus, it is imperative that the data partitioning scheme used for the arrays in spenta subroutines should expose parallelism along the principal dimensions, namely J and K, by removing the output dependences along those directions.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3.1: A 2D skewed distribution, indicating the processors that own the tiles.
This motivates the use of a skewed block distribution along the J and K dimensions for these arrays, as depicted in Figure 3.2.2. The I dimension is not tiled while the J and K dimensions are. The resulting blocks are assigned to the set of virtual processors using a cyclical scheme, resulting in a blocked local array. The arrays in the *spenta* solver of the J sweep, which are declared as \( \text{temp0}(I, J, \text{MAX}) \) and \( \text{wrk0}(I, \text{MAX}, J, \text{MAX}) \), would have their corresponding blocked local arrays declared as \( \text{temp}(I, \text{MAX}, n) \) and \( \text{wrk}(I, \text{MAX}, n) \), where \( n \) is the number of virtual processors. The last dimension numbers the blocks within each processor according to their tile number in the K dimension. Under this distribution, the principal loop can be tiled into a tiling loop that steps across the tiles in K and an intratile loop that steps within the elements of the tiles. The loop stepping across the J dimension would still be serial but the tiling loop can execute in parallel because the extra dimension corresponding to the block number within the processor serves to break the output dependences between the iterations of the principal K loop. Similarly, in the other two sweeps, the principal loop becomes parallelizable for the same reason.

When arrays other than those in *spenta* are computed from *tmp* and *wrk*, both the defined and used array elements must be colocated i.e. they must be mapped to the same processor to avoid unnecessary interprocessor communication. Given the structure of the computation in *fdl3di*, this implies that corresponding dimensions of all the arrays must be colocated. So, all the arrays must use the same partitioning scheme.

Since *fdl3di* is a stencil code, the computation of a tile \( t \) of an array may use elements from an adjacent tile of the same array, which is not owned by the processor.
that owns \( t \). Alternatively, it may use elements from a tile of another array which is adjacent to one owned by the current processor. In both these situations, it is necessary to communicate the needed elements from the processor owning the adjacent tile to the processor owning \( t \). This can be easily incorporated into the tiling scheme by expanding each tile to include a set of *shadow elements*, which contain data copied from other processors. For example, a stencil reference of the form \( \text{temp}0(i, j - 2) \) would require that each tile in the blocked local array \( \text{temp} \) have two shadow columns, with \( j \) values less than the lowest regular values, in which the data obtained by interprocessor communication is placed. Since the mapping scheme assigns adjacent tiles to adjacent virtual processors, this ensures that all communication is confined to neighboring virtual processors.

For an array \( a0(I\cdot\text{MAX}, J\cdot\text{MAX}, K\cdot\text{MAX}) \), the corresponding blocked local array would be declared as \( a(I\cdot\text{MAX}, J\cdot\text{MAX}/n, K\cdot\text{MAX}/n, n) \) (assuming there were no shadow elements). Thus, the storage per processor would decrease by a factor of \( n \). Hence, this data partitioning not only enables reduction in finish time by allowing the exploitation of parallelism and confining communication to neighboring processors, but also reduces the storage needed per processor.

None of the automated tools known to us, such as HPF, provides for a skewed block partitioning of the type described above and hence their use will limit the available parallelism. So, it becomes essential to use a manual approach to parallelize \texttt{fdl3di}. 
3.2.3 Elements of the Methodology

Manual parallelization of *fdl3d* was a challenging project considering the complexity of the code and the unfamiliarity of the project members with fluid dynamics. So, it was necessary to adopt an incremental methodology of parallelization that would allow a parallel program to be built subroutine by subroutine, with the output being validated after each such step. The methodology aims at creating a parallel program based on the Single Program Multiple Data (SPMD) programming model, as this ensures scalability and portability.

The task of the methodology is to translate the computation from a single address space to several decoupled address spaces. It is advantageous to retain the program in a sequential form executing in a single address space for as long as possible, as software development and debugging are more tractable in such an environment. This can be achieved by translating the data structures to an intermediate form which is similar to the blocked local form described earlier except that it tags each tile with the index of the virtual processor it is mapped to. For example, the array \( temp(1. JMAX, JMAX) \) would be translated into \( temp(1. JMAX, JMAX, n, n) \), where the elements of the last dimension signify processor indices and the other dimensions are as before. This intermediate form of the data structures, called the *blocked* form, makes it possible to write a sequential program that simulates the action of \( n \) virtual processors on these blocked arrays. Communication among the processors is simulated by copying the shadow elements between segments of an array that differ in the processor dimension. Subsequently, when building a truly parallel program, the bodies of the loops that run over processor indices can be enclosed in a conditional statement so that only the iteration in which the loop index matches the process identifier executes. Also,
the shadow copies mentioned above can then be replaced with explicit interprocessor communication of the shadow values. Thus, such a blocked program offers the benefits of a single address space but allows the testing and verification of issues relating to data partitioning and also makes the synthesis of the parallel program easier. So, the methodology operates in two passes, with the first (or the blocking) pass building a sequential program that manipulates the blocked arrays and the second (or the MPI) pass building a parallel program that operates on partitioned address spaces.

3.2.4 Building a Blocked Program

The conversion from the original program to the blocked form and then to the parallel form needs to be done in incremental steps so as to enable early and quick identification of errors that may be introduced during the conversion. Furthermore, the original sequential code and the arrays it manipulates are retained throughout the blocking pass and all but the final step of the MPI pass so that the output of the original code can be compared with that of the converted code at each incremental step. These aspects of the methodology are elaborated upon in this subsection.

Since fdl3di has a large number of subroutines, converting the program to a blocked form on a subroutine-by-subroutine basis is an attractive approach that not only allows incremental progress but also adapts well to team effort. Thus, the blocked program is an augmented sequential program in which each of the original subroutines has been augmented with a counterpart that simulates the action of several virtual processors on the blocked arrays. The arrays computed by the sequential routine but used outside it are compared element by element with the corresponding blocked arrays. Even though the arrays store floating point values, they were compared for
exact equality rather than equality within a tolerance factor, because exact match
between the sequential and parallel output was one of the project goals.

Before a subroutine can be converted to operate on the blocked arrays, all the sub-
routines that it calls must already have been converted. This necessitates a bottom-up
traversal of the call tree of the program, in which the leaf nodes are converted before
the nodes that call them. Since *fdl3di* has no recursive subroutines, the structure
of subroutine calls is a tree rather than a graph, thus making bottom-up traversal a
well-defined concept. The conversion of each node includes not only the generation
of a blocked routine but also the generation of an initialization routine which copies
data values from the arrays used by the original subroutine to the blocked arrays
used by the blocked routine. Thus, the call to each sequential subroutine *aaa* with
arbitrary arguments is transformed into a set of 4 calls as follows.

```
call aaa_init

call aaa(xx, yy)    \implies\     call aaa(xx, yy)
call aaa_b(xx', yy')
call aaa_cmp(locator)
```

Here, *aaa_init* is the initialization routine. *aaa_b* is the blocked routine and *aaa_cmp*
is the compare routine. For scalar arguments, the argument *xx' to aaa_b* is identical
with the corresponding one *xx* for *aaa*. However, for array arguments *yy* that are
passed to *aaa*, the corresponding argument to *aaa_b* is the equivalent blocked array.

Since this set of routines could be invoked within a loop or at multiple points in the
program, the routine *aaa_cmp* is given an integer argument *locator* to identify, in the
event of a discrepancy between the computations of *aaa* and *aaa_b* which invocation
of *aaa_b* generated the discrepancy.
Since the call tree is traversed bottom-up, all the routines called in a given routine, say aaa, will be blocked before aaa itself is blocked. At that point, aaa can be blocked to yield aaa.b. The calls to individual initializations of the called routines are deleted and a new init routine aaa.init is written which copies only the arrays that are used within aaa before being defined. A new compare routine aaa.cmp is written and the call to aaa is then transformed into a set of 4 calls, as explained above. Eventually, the entire program will have only a single init. to initialize the blocked equivalents of the input arrays. It is desirable to retain more than a single compare call since the differences reported by them help in determining the location of errors in the blocked code.

It is useful to build a table that lists, for each routine, the arrays defined and used by that routine. This helps in determining which arrays are to be copied in an init routine (namely, the ones used before being defined in the sequential routine) and which arrays are to be compared in a compare routine (i.e. the arrays that are defined in this routine and used or output subsequently). It is also an invaluable aid in debugging and performance tuning.

The rest of this subsection examines the issues involved in building a blocked counterpart of a sequential subroutine.

**Building a Blocked Routine**

A blocked routine performs the same actions on tiled arrays that the corresponding sequential routine performs on the equivalent original arrays. It essentially simulates the actions of a set of processors on the data they own. Thus, the aim of a blocked routine is to distribute, among a set of virtual processors, the iterations of each loop that is used to index a tiled dimension of an array. In principle, this involves tiling each
loop that indexes a tiled dimension of an array into two loops: a tiling loop that steps across the tiles in that dimension and an intratile loop that steps across the elements in each tile. (The loop bounds of the intratile loop may depend on the tile index that is varied by the tiling loop.) The tiles in the iteration space need to be assigned to processors in such a way that the processor which executes a specific iteration owns all the array elements accessed by that iteration i.e. the "owner computes" rule is followed. In other words, since the objective is to build a data parallel program, the data partitioning is used to induce a task partitioning. The issues that arise in the course of task partitioning are discussed now.

A loop nest in the final parallel program accesses only those tiles of the blocked local arrays which have been mapped to the processor that executes the loop nest. Hence its precursor in the intermediate blocked program must execute the loop nest for each value of the processor index. Since the blocked arrays, unlike the blocked local arrays, have a processor dimension, such a repeated execution of the loop nest constitutes a simulation of a distributed address space program in a single address space. This simulation must be done in such a way that it is easy to convert it into a fully parallel form. For a loop nest in the sequential program, all of whose loops are parallelizable, this is achieved by enclosing the corresponding blocked loop nest in an outer loop that steps over the range of processor indices. Subsequently, when a parallel program needs to be built from the blocked program in the MPI pass, the body of the processor loop can be enclosed in a conditional statement that is executed only when the loop index matches the processor identifier of the MPI program.

However, if the sequential loop nest includes loops that step across both the J and K dimensions of an array, tile indices along these two dimensions together specify a
owning processor for the array’s tile in data space, which may not match the loop counter variable of the processor loop. To avoid this conflict, the innermost among the J and K loops, say J, has its tiling loop removed and its tile index calculated from the index of the processor loop and the tile index of the K loop. This is made possible by the fact that, since each tiled array dimension is distributed across all the processors, the combination of a processor index and one of the tile indices will uniquely determine the other tile index. The intratile loop of J, whose bounds may depend on the tile index along J, is unaltered. Thus, in general, all but one of J and K loops in a loop nest is tiled to yield a tiling loop and an intratile loop, while the innermost among them is converted only into an intratile loop.

While most loop nests in fdl3di can be converted in this way, the loop nest constituting spenta is not amenable to this approach in the J and K sweeps. This is because a loop that carries a data dependence and indexes a tiled dimension of an array, such as the loop that indexes the sweep dimension in the spenta of J and K sweeps, cannot be parallelized. While it may be tiled, its iterations must be executed in tile order whereas enclosing it in a processor loop would execute it in the order of processor numbering. Furthermore, the principal loop surrounding such a loop, although parallelizable, cannot be enclosed by a processor loop. In these two cases, the processor index is calculated from the tile indices along J and K. Note that it is still possible to convert such a blocked loop nest to a fully parallel form by employing conditional execution as described above.

Shadow copies

As discussed earlier, shadow copies in a blocked program simulate the interprocessor communication in an MPI program. A shadow copy is a loop nest that contains
a set of assignment statements, each referring to a single array, with the destination being a shadow element and the source being an array element that resides on a processor adjacent to the one owning the shadow element. The MPI pass replaces them with communication based on calls to the MPI library. Hence, the principal factors that reduce communication must be addressed in the placement and design of the shadow copies.

To transfer as large an amount of data as possible in each communication, it is necessary to place the shadow copies at the loop level of the lowest possible depth within its loop nest. This implies a position at the highest loop level that is common to the stencil reference that needs the shadow element and the reaching definition that defines its source. By appropriate use of loop fusion, shadow copies of several arrays can be placed in a single loop nest, which will be mapped to a single MPI communication in the final pass.

3.2.5 Building the MPI Program

The blocking pass yields a sequential program with both the original code and the blocked code that produces the same output as the original. This needs to be converted into a parallel program in which each processor accesses only its local data. The shadow copies are replaced by communication across processors. These communications also serve to synchronize processors that handle the iterations of loops that carry dependences, such as the sweep loop in spenta. This is accomplished by the MPI pass in three stages, again in an incremental manner. These stages are discussed further in this subsection.
The first stage of the MPI pass begins by building a parallel program that essentially executes the sequential version built by the blocking pass on many processors. This is done by adding a call to the MPI initialization routine MPI_INIT() at the beginning of the program and a call to the MPI termination routine MPI_Finalize at the end. Thus, all computations are completely replicated among the processors and the shadow copies are still retained. Then, each of the shadow copies is supplemented with a MPI communication. That is, following a shadow copy from source processor \(pe'\) to sink processor \(pe\), a communication is placed in which the processor with \(myid = pe'\) sends a set of array elements and the processor with \(myid = pe\) receives it and updates its shadow elements. However, the shadow copies are performed for all values of the \(pe\) index while the communication-based shadow update is performed only for \(pe = myid\). This is so because each processor needs to carry out the blocked routine computations for all values of \(pe\) until all interprocessor communication is in place. Another point of note is that the MPI communication is placed after the shadow copy so that it overwrites some of the shadow elements that were updated by the shadow copy. This makes it likely that errors in the communication will affect the subsequent computations and thus manifest as differences reported by the compare routines. However, it does not expose certain categories of errors, such as message sizes being off by one. When all interprocessor communication is in place, the first stage is complete.

The second stage removes the replication of computations. This is done by conditionally executing the body of each \(pe\) loop, except those involved in MPI communication, only for \(pe = myid\). This renders all shadow copies invalid because the source elements of the shadow copy, which reside on a different processor, will not be
computed correctly. So, all the shadow copies can be commented out. Note that conditional execution of even a single computational loop potentially renders all shadow copies invalid, so that the correctness of all the MPI communication is tested by this change. Thus, while the changes in the code are always done incrementally, testing is not incremental at this stage of the methodology. This stage results in a parallel program with purely local computations that uses interprocessor communication to update the shadow elements. However, it still retains the original sequential code and the compare routines.

The third stage eliminates the processor dimension of all the arrays since each processor accesses only one element in it. This can be done by a global replacement of the last subscript of all array references and the last dimension of all array declarations. This stage also eliminates all the sequential code and the compare routines, resulting in a purely parallel program.

### 3.3 General Formulation of the Methodology

While the methodology has been developed for easing manual parallelization of a single application, its approach can be generalized to many computations that share the stencil structure of $f_{dl3di}$ and it can also be automated because each of its stages involves a well-defined code transformation. An automated form of the methodology may be expected to do better than general purpose tools for automatic parallelization since it focuses on a large but well-defined class of computations and can thus address performance issues more optimally. Furthermore, if the blocking pass and the MPI pass were implemented by separate tools, then several data distributions could be investigated easily by studying the performance profile of the blocked sequential code.
The ability to investigate data partitioning issues in a shared address space is a unique feature of the methodology. The rest of this section abstracts the issues that arose in the case study to formulate a general methodology for a large class of stencil codes and also presents the code transformations needed in each stage of the methodology in an algorithmic form that lends itself to automation.

### 3.3.1 Domain of the Methodology

The class of applications that can be handled by the methodology need to have a stencil structure since the ability to represent all interprocessor as copying of shadow elements relies on it.

The domain of the methodology is the set of nonrecursive stencil programs in which each array subscript is either:

- **C1**: an integer, or
- **C2a**: a loop variable, if the subscript occurs in a reference that defines an array, or
- **C2b**: of the form $i \pm s$, where $i$ is a loop variable and $s$ is an integer, if the subscript occurs in an array use reference.

The compliance of a program with these conditions can be verified mechanically. Furthermore, from condition C2(b), the stencil size of any array in a given dimension can be computed as the largest among the $s$ values in that dimension: the largest $s$ occurring in a subscript of the form $i + s$ is the positive stencil while the largest $s$ in a subscript of the form $i - s$ is the negative stencil.
The program is viewed as a collection of routines, each with a sequence of loop nests. A loop nest is defined as a set of nested loops and statements within a subroutine. Loop nests which are not enclosed by other loops in the same subroutine are said to be maximal. These statements could be assignments, input/output, calls to subroutines or conditional constructs. All loops are converted to be of unit stride, as part of the methodology. This, together with condition $C_2$, ensures that all arrays are accessed with unit stride. Also, the program is assumed to be well-structured i.e. its control flow is expressed in terms of sequence, conditional and loop constructs only.

3.3.2 Three Pass Structure

The methodology broadly follows the procedure adopted to convert $fdl3di$ in the case study. However, since the methodology is aimed at a broad class of applications, the procedure needs to be expanded in several ways. Firstly, before the methodology can be applied, it is necessary to confirm that the target program conforms to the criteria listed above. Secondly, the data distribution in the methodology is chosen from a general class of tiled distributions, as a generalization from the case study. The chosen data distribution must be such that the elements of an array that are owned by a processor $p$ should be computable from elements of the same or other arrays that are also owned by $p$, i.e., arrays should be colocated. While issues regarding the choice of a data distribution and colocation are discussed in greater detail in Section 3.3.4, the salient point to note is that data colocation must be verified before the methodology is applied.

Thirdly, the algorithmic code transformations employed in the methodology assume the availability of information obtained by control flow and data flow analysis.
of the input program. For example, whether the index variable of a given loop is used to subscript a tiled dimension of an array is of relevance in transforming that loop into a blocked form. Also, the code transformations assume that the program is cast in a normalized form that simplifies the algorithms. If the input program is not in such a form, it needs to be converted into such a form. Details regarding the normalized form are presented later. These preparatory transformations need to be applied before the program is ready for further processing.

This discussion indicates that, while the conversion of *fdl3di* was structured in two passes, in the general methodology, there needs to be a preliminary pass to perform the following tasks: verify that the given program conforms to the criteria of the methodology, allow the user to choose a data distribution, verify that the data distribution chosen leads to colocation of arrays and, finally, perform the set of program analyses and transformations needed later. The other two passes, the blocking pass and the MPI pass, are suitably generalized to handle all applications in the target domain.

### 3.3.3 A Running Example

To illustrate the concepts and application of the methodology, the program fragment in Figure 3.2 will be used as a running example throughout the paper.

The code implements Jacobi’s algorithm for solving a system of linear equations. It uses two 2D arrays, called *a0* and *b0*, both of size \( M \times M \), where \( M \) is a compile-time constant. Arrays in the original program have names that conventionally end in `'O'` while corresponding tiled arrays do not. The parameter *NumIter*, which controls the number of iterations of the algorithm, is also a compile-time constant.
real a0(M,M), b0(M,M)

do k = 1, NumIter
    do j = 2, M-1
        do i = 2, M-1
            a0(i,j) = b0(i-1,j) + b0(i,j-1) + b0(i+1,j) + b0(i,j+1)
        enddo
    enddo
    do j = 2, M-1
        do i = 2, M-1
            b0(i,j) = a0(i,j)
        enddo
    enddo
enddo

Figure 3.2: A running example implementing an equation solver using Jacobi.

When the methodology is applied to this program, the preliminary pass first ensures that this code falls in the domain of the methodology. Other elements of this pass are discussed in the next subsection.

3.3.4 Data Distribution

The natural class of data distributions for stencil codes, the target of the methodology, are tiled distributions, such as block, cyclic or skewed distributions. The choice of a distribution affects performance strongly and is difficult to automate. So, the user needs to choose a tiled data distribution for each dimension of every array in the program and also the mapping of the tiles to a pool of virtual processors. Since each processor follows the 'owner computes' rule, the primary objective of this step is to ensure colocation i.e. the elements of an array that are owned by a processor $p$ should be computable from elements of the same or other arrays that are also owned by $p$. In
other words. for an array \( a_0 \) that is defined using an array \( b_0 \), some dimensions of \( a_0 \) would need to be tiled with the same distribution and mapping as certain dimensions of \( b_0 \), so that these dimensions are collocated. For any array dimension, the set of dimensions in other arrays that it is collocated with can change from one subroutine to another due to changes in that array's access pattern, as discussed below. Array dimensions that cannot be distributed without ensuring colocation are replicated across the processors. In particular, since scalars are viewed as 0-dimensional arrays, they are always replicated. (During the blocking pass, since the program is sequential, the replication is not actually done but a single copy is used.) Another objective of the data decomposition is the maximization of load balance across the processors i.e. the arrays should be tiled in such a way that each processor receives roughly equal computational load. This is achieved by requiring that each tiled dimension of an array be distributed among all the processors. The mappings are constant throughout the program i.e. there is no redistribution of any array in the generated parallel program. This section describes the abstraction of the data distribution and mapping, algorithmic conversion of array declarations and references into a tiled form, changes in the view of an array and the formal verification of colocation.

Consider a \( \Delta \)-dimensional array \( a_0 \) in which \( d \) dimensions are to be tiled to yield a blocked array \( a \). Dimension \( k \) is split into \( n t_k \) tiles and is thus converted to two dimensions: a tile dimension, that steps across the tiles, and an intratile dimension that steps across the elements within a tile. This distribution is abstracted using a set of functions that may be implemented as arrays or as function subprograms. Since the size of each tile in \( a_k \) (the \( k \)-th dimension of \( a \)) will not necessarily be the same, a tile size function \( ts_k : \mathcal{N} \to \mathcal{N} \) is required which takes a tile index \( t_{ik} \) in \( a_k \)
and returns its extent. The range of the elements within a tile is \( \{0, \ldots, ts_k^a(ti_k) - 1\} \) by convention. The smallest among the tile sizes should exceed the stencil size in that dimension, for reasons to be explained later. It is also useful to have a locator function \( tile - elem_k^a : \mathcal{N} \rightarrow \mathcal{N}^2 \) which converts an index in the \( k \)-th dimension of \( a_0 \) into a tile-element pair corresponding to that index. If the integer is out of range, the tile number returned is \(-1\) if the integer is less than the lowest index in \( a_k \) and \( n_{t_k}^a + 1 \) if it exceeds the highest index. The tiling of the \( d \) dimensions of \( a_0 \) generates a decomposition of \( a_0 \) into blocks. The mapping of a block onto a unique processor is abstracted by a function \( procnum^a : \mathcal{N}^d \rightarrow \mathcal{N} \) which takes a set of \( d \) tile indices that describe a block and returns the index of the processor to which that tile is mapped. The blocks within a processor are organized linearly and addressed using a **processor block number**, which is computed by a function \( peblk^a : \mathcal{N}^d \rightarrow \mathcal{N} \) that takes a set of \( d \) tile indices as input arguments. Some tiled distributions (such as block-cyclic ones) assign more than one tile from a given dimension of an array to the same processor. These tiles are also organized linearly and numbered with a **dimension block number**, which is computed by a function \( dimblk_k^a : \mathcal{N} \rightarrow \mathcal{N} \) which takes a tile index in \( a_k \) as input. The number of blocks of an array that are mapped to the same processor is given by \( nblk^a : \mathcal{N} \rightarrow \mathcal{N} \) which takes a processor number \( pe \) as its argument, while the number of tiles of a dimension \( k \) that is assigned to a processor is given by \( ndblk_k^a : \mathcal{N} \rightarrow \mathcal{N} \) which also takes a processor number \( pe \) as its argument. These functions are so chosen that it is possible to design inverse functions \( tile_k^a : \mathcal{N}^2 \times \mathcal{N}^{d-1} \rightarrow \mathcal{N} \) that calculate the tile index in \( a_k \) given the processor number, the dimension block number and the other \( d - 1 \) tile indices.
In general, these functions are array-specific. But, when two dimensions of two arrays are colocated, the distribution and mapping of those dimensions need to be identical so that these functions will also be identical for those dimensions, except that the value of \( d \) may be different for them. Finally, the colocation of dimensions in two arrays \( a \) and \( b \) is modeled by the function \( cl^{ab} : \mathcal{N}^2 \rightarrow \mathcal{N} \), which takes as input a subroutine identifier and the number of a dimension in \( a \) and returns the number of the dimension in \( b \) that is colocated with that dimension of \( a \) in that routine, if any, or zero, otherwise. This assumes that the subroutines in the original program have been numbered uniquely.

Array references can be translated into a blocked form in an algorithmic manner since, by condition C2, they can take only two forms. Figure 3.3 presents the algorithm that takes an arbitrary array reference and translates that into a blocked form. The transformation of subscripts in tiled dimensions done in algorithm ConvertSubscript. Since colocation is ensured in each assignment statement, the processor that owns the array element \( e \) being defined must also own the array elements used to compute \( e \). Thus, the conversion of a subscript \( \sigma_k \) in a tiled dimension \( a_k \) of a use reference must take into account the possibility that a dimension of the array \( b \) being defined is colocated with \( a_k \). Note that, if an array \( a \) is used to compute an array \( b \) and the \( k \)-th subscript \( \sigma \) of \( a \) is of the form \( i \pm s \), then a dimension of \( b \) colocated with \( a_k \) has to have \( i \) as its subscript, to ensure colocation. Also, in the case of integer subscripts, the array reference has to be protected by a guard that matches the current tile index (which may be varied in a loop) against the index of the particular tile containing that integer. Other references in the statement cannot possibly be invalidated by the guard even if they involve a colocated dimension.
algorithm GenBlockedReference(r)
given reference $r = a_0(\sigma_1, \ldots, \sigma_\Delta)$ of array $a_0$ conforming to condition C2
output the blocked reference $r_b = a(\tau_1, \ldots, \tau_\Delta, nb, pe)$ of $a$ in which subscripts of
tiled dimensions range only over a tile extent. The processor number $pe$
and the processor block number $nb$ are assumed to be available at
the statement containing $r$.

for $\lambda \leftarrow 1, \Delta$ do
    if $\lambda$ is not a tiled dimension then
        $\tau_\lambda = \sigma_\lambda$
    else
        $\tau_\lambda = \text{ConvertSubscript}(r, \lambda, \sigma_\lambda)$
    endif
endfor
$r_b \leftarrow a(\tau_1, \ldots, \tau_\Delta, nb, pe)$
end

algorithm ConvertSubscript($r_a, k, \sigma$)
given reference $r_a$ of array $a$, a tiled dimension $k$, $k$-th subscript $\sigma$ of $r_a$
output the $k$-th subscript $\tau$ of the blocked reference
a guard to validate the blocked reference, if necessary
Notation $a_k$: $k$-th dimension of $a$
$ts_k^a(ti)$: tile size of the tile numbered $ti$ in $a_k$

if $\sigma$ is an integer then
    if $r_a$ is a use and array $b$ defined using $r$ has a dimension $b_l$
colocated with $a_k$ then
        {The $l$-th subscript in $r_b$. the define of $b$. has to be an integer. say $\gamma$.}
generate code to compute $(t_\gamma, l_\gamma) = \text{tile-elem}^b_k(\gamma)$
generate code to compute $\tau = l_\gamma + \sigma - \gamma$
else
    generate code to compute $(t_\sigma, \tau) = \text{tile-elem}^a_k(\sigma)$
    if $r_a$ is a define or $r_a$ is the first use reference in its statement then
        {Let $ti_k^a$ be the index of current tile in $a_k$ processed}
        {In the current loop nest.}
        generate an IF-THEN guard around the statement that checks $ti_k^a = l_\sigma$
    endif
endif
else
    $\sigma \equiv i \pm s$, where $i$ is a loop variable and $s$ is an integer.
    {For a reference that defines an array, $s$ will be zero.}
generate code to compute $(ti_k^a, li) = \text{tile-elem}^a_k(i)$
generate code to compute $\tau = li \pm s$
endif
end

Figure 3.3: Algorithm to convert a given array reference to a blocked form.
Note that a subscript of the form \( i_k \pm s \) gets translated to \( li_k \pm s \), where \( li_k \) is the modulus of \( i_k \) with respect to the tile size. However, the subscript \( li_k \pm s \) will access elements outside the tile in general. For example, in the example of Figure 3.2 with \( M = 25 \), the array \( b(0(25.25)) \) could be tiled along the first dimension to yield 5 tiles of 5 elements each. Converting a reference of the form \( b(0(i - 1.j)) \ (i,j \in \{2, \ldots, 25\}) \) into \( b(li - 1.j) \ (li \in \{2, \ldots, 5\} \) for the first tile and \( li \in \{1, \ldots, 5\} \) for others) results in an access to \( b(0.j) \) for all except the first tile, which is outside the tile. Since each tile is owned by a processor, this may require an off-processor access. Thus, the tiles need to be augmented with a boundary column \( b(0.1 : 25) \) which contains a copy of \( b(5.1 : 25) \) of the preceding tile. In general, stencil references of the form \( i_k + s_k^p \) and \( i_k - s_k^n \) in a \( \Delta \)-dimensional array will require \( s_k^p \Delta - 1 \)-dimensional boundary planes in the direction of increasing \( k \) and \( s_k^n \Delta - 1 \)-dimensional boundary planes in the direction of decreasing \( k \). These boundary elements are called shadow elements and represent a generalization of the shadow elements used in \textit{fdl3di}. There could be shadow planes in many tiled dimensions.

The shadow elements need to be assigned values by moving data within the array. In the first pass of the conversion, which retains the sequential nature of the program, this is accomplished by explicit copying into the shadow values. In the second pass, when a parallel MPI-based program is built, the assignment is done by communication among the processors. Since, for any array, the tile size in any dimension exceeds the stencil size in that dimension, the shadow elements will always contain copies of elements from only an adjacent tile. (Tiles \( p \) and \( q \) are said to be adjacent if there are elements \( a \in p \) and \( b \in q \) whose coordinates differ only by zero or unity in every dimension.)
The generation of declarations for the blocked arrays can also be formally specified. Consider the case where no shadow elements are involved. The original array declaration $a(\text{lim}_1, \text{lim}_2, \ldots, \text{lim}_\Delta)$ would be transformed into a declaration of a tiled array as $a(\text{loc}_1, \text{loc}_2, \ldots, \text{loc}_d, \text{lim}_{d+1}, \ldots, \text{lim}_\Delta, \text{nblk}, \text{npe})$, where $\text{lim}_k$ are the ranges of the global indices, $\text{loc}_k = \max_k \{t\text{s}_k(ti_k)\}$ are the ranges of the local indices, $\text{nblk}$ is the number of local blocks in each processor and $\text{npe}$ is the number of processors. Each processor updates only one out of the $\text{npe}$ elements in the last dimension. However, since the blocking pass builds a sequential program which simulates the action of several processors, it is possible for one 'processor' to directly access the elements of another.

If the $k$-th dimension $a_k$ is tiled and has a stencil of size $s_k^p$ in the increasing direction and $s_k^m$ in the decreasing direction, the array will need shadow values. This is handled by translating the original declaration into $a(-s_1^m + 1 : \text{loc}_1 + s_1^p, -s_2^m + 1 : \text{loc}_2 + s_2^p, \ldots, -s_d^m + 1 : \text{loc}_d + s_d^p, \text{lim}_{d+1}, \ldots, \text{lim}_\Delta, \text{nblk}, \text{npe})$. Here, the 'normal' range of the local index in the $k$-th tiled dimension is $\{1, \ldots, \text{loc}_k\}$ while the ranges $-s_k^m : 0$ and $\text{loc}_k + 1 : \text{loc}_k + s_k^p$ are the shadow points. Note that shadow columns along two different dimensions intersect. This allows access to adjacent tiles that differ in more than one dimension.
Consider the array \( b0(M,M) \) in the example of Figure 3.2. It has a stencil size of 1 in both its dimensions. Since the second dimension is indexed by the outermost parallelizable loop, a good choice of a data distribution is a blocking along that dimension. The blocked equivalent will be declared as \( b(M, 0 : \frac{M}{n} + 1 : 1, n) \), where \( n \) is a parameter representing the number of processors. The number of blocks of \( b \) per processor is one, as indicated by the range of the dimension before the last, which stores the processor block number. A possible choice of the needed functions can be expressed in terms of the tile index \( jn \) along the second dimension, as:

\[
\begin{align*}
\text{tile} - \text{elem}_i^{b}(\lambda) &= \lambda \mod \frac{M}{n} \quad \text{for } 1 \leq \lambda \leq M. \\
\text{procnum}^{b}(jn) &= jn. \\
\text{peblk}^{b}(jn) &= 1. \\
\text{dimblk}^{b}(jn) &= 1. \\
\text{nblk}^{b}(pe) &= 1. \\
\text{tile}^{b}(pe) &= pe.
\end{align*}
\]

where all the free variables range from 1 through \( n \). Note that the tile size functions return the `normal' extent of a tile, not its extent with shadows. The declaration of both the arrays in the running example under the chosen blocked distribution is shown in Figure 3.4. Note that their second dimensions are colocated.

Consider a program in which an array \( a0(100) \) is computed from another array \( b0(100 : 100) \) as row-wise or column-wise sums i.e. \( a0(i) \) contains the sum of \( b0(i : 100) \) in one routine of the program and the sum of \( b0(100 : i) \) in another routine. In the corresponding blocked program, it is then necessary to colocate the tiled array \( a \) with the first dimension of \( b \) in some routines and with its second dimension in others. If the two dimensions of \( b \) are not distributed identically, the access pattern of \( a \) changes between the routines. This access pattern is also called a view. Thus, the colocation function \( c_t^{ab} \) needs to change when the view changes. The methodology allows view changes at the granularity of subroutines.
The colocation criterion requires that the elements of an array \( a \) that are owned by a processor \( p \) should be computable from elements of the same or other arrays \( b \) that are also owned by \( p \). In terms of array dimensions, this translates to selecting the same distribution and mapping for certain dimensions of \( a \) and \( b \), so that their tiles are in one-to-one correspondence. But, in an assignment statement with a definition reference of \( a \) and a use reference of \( b \), a subscript of \( a \) is restricted to be a loop variable \( i \) (if not a constant) while the subscript in the colocated dimension of \( b \) can be of the form \( i \pm s \), where \( i \) is a loop variable and \( s \) is an integer. So, the tiles that these elements belong to may not be owned by the same processor. However, the tiles of \( b \) can be extended with shadow elements that contain copies of elements from adjacent tiles, as discussed. With the addition of shadow elements, the reference to \( b \) can refer to a tile that is owned by \( p \). Thus, the requirement of colocation incorporates the notion of shadow elements, in general.

The methodology provides algorithmic support for verifying colocation i.e. checking whether a given program can be parallelized under a given set of array distributions. The problem of verifying colocation for the entire program consists of demonstrating for every pair of array references involved in a definition-use relationship that the array element being used resides on the same processor as the array element being defined. This demonstration is done using the algorithm \texttt{VerifyColocation}, presented in Figure 3.5. The algorithm makes use of the following reasoning. Since the only permissible subscripts for \( a \) are integers or loop variables, and those for \( b \) are integers or of the form \( i \pm s \) (where \( i \) is a loop variable and \( s \) is an integer), the difference between subscripts in colocated dimensions of \( a \) and \( b \) can only be an integer. If it is not, or if the integer difference exceeds the number of shadow elements in the
algorithm VerifyColocation
given a pair of array references \( a(\sigma^a_1, \ldots, \sigma^a_{\Delta_a}) \) which is defined using \( b(\sigma^b_1, \ldots, \sigma^b_{\Delta_b}) \)
• a set of loop variables \( \{i_1, \ldots, i_m\} \) and their corresponding loop bounds. \( \{l_1 : u_1, \ldots, l_m : u_m\} \), representing the set of loops that bound the pair of references.
output an error message if the two arrays are not truly colocated. none otherwise

for \( n \leftarrow 1, \Delta_a \)
\( d \leftarrow cl^{ab}(n) \)
if \( d > 0 \) then \( \{b_d \text{ is colocated with } a_n\} \)
\( \{\text{Let } \tau \text{ denote the difference } \sigma^b_d - \sigma^a_n. \text{ Let } s^+ \text{ and } s^- \text{ be the}\} \)
\( \{\text{stencil sizes in the increasing and decreasing direction of } b_d, \text{ respectively.}\} \)
if \( \tau \) is not an integer then
write "a and b not colocated"
else if \( \tau > 0 \) and \( \tau > s^+ \)
write "a and b not colocated"
else if \( \tau < 0 \) and \( -\tau > s^- \)
write "a and b not colocated"
endif
endif
endfor
end

Figure 3.5: Algorithm to verify that a given data distribution scheme satisfies the colocation criterion.

appropriate (positive or negative) direction of \( b_d \), the colocation criterion would be violated.

3.3.5 The Blocking Pass

The overall structure of the blocking pass is identical with that in the case study but is recapitulated here for completeness. Since the given program is nonrecursive, it is possible to build a call tree of the subroutines called in the program. The blocking
pass traverses the call tree in a bottom-up manner. At each node, the sequential routine at that node, say \texttt{aaa}, is augmented with three other routines: a blocked routine \texttt{aaa.b}, an initialization routine \texttt{aaa.init} and a compare routine \texttt{aaa.cmp}.

The blocked routine \texttt{aaa.b} performs the same actions on tiled arrays that the original routine \texttt{aaa} performs on the corresponding original arrays. It essentially simulates the actions of a set of \(npe\) processors on the data they own. Interprocessor communication is simulated with shadow copies. The algorithm to build a blocked routine from the original is presented in Section 3.2.4. The initialization routine \texttt{aaa.init} copies the arrays that are used by the sequential routine before being defined into the corresponding blocked arrays. The compare routine \texttt{aaa.cmp} compares the arrays calculated by \texttt{aaa} that are used outside \texttt{aaa} with the corresponding arrays in the blocked routine. It may compare the shadow elements of the arrays also. Should the comparison fail, it writes out the name of the array and the subscripts where it differed. Since there may be many calls to \texttt{aaa}, or it may be called in a loop, \texttt{aaa.cmp} is passed a locator argument, which is an integer that identifies the particular invocation of the compare. Thus, the call to each sequential routine with arbitrary arguments is transformed into a set of 4 calls as follows.

\[
\text{call } \texttt{aaa.init} \\
\text{call } \texttt{aaa(xx, yy)} \quad \Rightarrow \quad \text{call } \texttt{aaa(xx, yy)} \\
\quad \text{call } \texttt{aaa.b(xx', yy')} \\
\quad \text{call } \texttt{aaa.cmp(locator)}
\]

where \(xx\) and \(yy\) could be scalar or array arguments. For scalar arguments, the argument \(xx'\) to \texttt{aaa.b} is identical with the corresponding one \(xx\) for \texttt{aaa}. However,
for array arguments \( y \) that are passed to \( \text{aaa} \). the corresponding argument to \( \text{aaa}_b \)
is the equivalent blocked array.

Since the call tree is traversed bottom-up. all the routines called in a given routine. say \( \text{aaa} \). will be blocked before \( \text{aaa} \) itself is blocked. At that point. \( \text{aaa} \) can be blocked to yield \( \text{aaa}_b \). The calls to individual initializations of the called routines are deleted and a new init routine \( \text{aaa}_\text{init} \) is written which copies only the arrays that are used within \( \text{aaa} \) before being defined. A new compare routine \( \text{aaa}_\text{cmp} \) is written and the call to \( \text{aaa} \) is then transformed into a set of 4 calls. as explained above. Eventually. the entire program will have only a single init. to initialize the blocked equivalents of the input arrays. It is desirable to retain more than a single compare call since the differences reported by them help in determining the location of errors in the blocked code.

**Building a Blocked Routine**

The basic approach adopted to convert a routine to its blocked form is the same as in the case study. The two essential points of this approach are that. wherever possible. loop nests are enclosed by a loop that steps over the range of processors and that. all but one of the blockable loops in the nest are tiled to yield a tiling loop and an intratile loop. with the innermost blockable loop transformed only into an intratile loop. its tile index being calculated from the index of the processor loop and the rest of the tile indices. The latter feature was motivated by the observation that. since each tiled dimension of each array is distributed across all the processors. the combination of the processor index with all but one tile index uniquely determines the last tile index. However. as noted in Section 3.2.4, iterations of a blockable loop that carries a dependence cannot be executed in parallel. i.e.. its iterations must be executed in tile
order rather than processor order so that enclosing such a loop in a processor loop is impermissible. Such a loop can be tiled as before and can enclose a processor loop if it does not enclose other blockable loops with loop-carried dependences (LCDs). If the innermost blockable loop in a loop nest carries a dependence, as was the case in *spenta* in the case study, it is not possible to have a processor loop at all, but the processor index can be calculated from the set of tile indices. These considerations motivate the need to transform loops differently based on whether or not they carry a dependence. This subsection presents algorithms to translate a sequential loop nest into a blocked form in a manner that preserves dependences and yet exploits all the parallelism available.

Consider a loop nest L in which none of the loops has a LCD and which does not have any subroutine calls in its body. The mechanism of converting such a loop nest is presented in Figure 3.6 as the algorithm `BasicConvert`. The algorithm, like the other algorithms in the methodology, is formulated in terms of converting the programming constructs involved in the loop nest, i.e. sequential, conditional and loop constructs. The main `foreach` loop that steps over the constructs focuses on blockable loops, treating those that enclose other blockable loops differently from the innermost blockable loop, as discussed earlier. The task of tiling non-innermost blockable loops into a tiling loop and an intratile loop is handled by a subroutine called `GenTilingLoops`. Each blockable loop $i_k$ in the original loop nest gives rise to an intratile loop $li_k$, whose range depends in general on the index $ti_k$ of the current tile. (Since the tile extents of the colocated dimensions of all arrays are identical, the tile index of any one of them may be used.) The bounds are calculated using the algorithm `GetBounds` from the original loop bounds, which may be constants or
algorithm BasicConvert(L)
given a loop nest, with nest number L, whose loops have loop variables \{i_1, \ldots, i_n\}, lower bounds \{Lb_1, \ldots, Lb_n\} and upper bounds \{Ub_1, \ldots, Ub_n\}, where each bound may be an integer or a function of outer loop variables. tiled(L) is the set of blockable loops in the nest and its cardinality is \(n_d(L)\). The control constructs forming the nest is given by the set \(\Sigma\). None of the loops have loop-carried dependences or subroutine calls. Blockable loops in a calling routine that enclose this nest are given as the possibly empty list \(I_1, \ldots, I_m\).

output transformed loop nest \(L_b\) in which loops that correspond to tiled dimensions in any array are blocked to yield new bounds \(\{lb_1, \ldots, lb_n\}\) and \(\{ub_1, \ldots, ub_n\}\).

foreach integer subscript \(s\) of any array \(a\) in a tiled dimension \(a_k\) in \(L\) do
  generate the tile index in \(a_k\) using \(tile^{\Sigma}(s)\)
endfor
for \(k \leftarrow 1, m\) do
  generate the tile index corresponding to the loop variable \(I_k\)
endfor
for \(k \leftarrow 1, n\) do \(\{\text{Compute tile bounds for loops in } L\}\)
  \((t_l_k, t_u_k, l_k, u_k) \leftarrow \text{GenTileBounds}(Lb_k, Ub_k)\)
endfor
generate the pe loop as “do pe = 1, npe”
foreach construct \(c\) in \(\Sigma\) do
  if \(c\) is a loop \(i_k\) then
    if \(i_k\) is not blockable then
      copy it as it is
    else if \(i_k\) encloses blockable loops then
      GenTilingLoops(i_k)
    else \(\{\text{it encloses only nonblockable loops and scalar constructs.}\}\)
      \{\text{Let } a \text{ be the first array reference in the body of } i_k \text{ to have a tiled dimension}\}
      \{\text{indexed using} i_k. \text{Let } a \text{ have } d_a \text{ tiled dimensions.}\}
      \{\text{whose tile indices are } t_1, \ldots, t_{d_a}\}
      \{\text{generate a loop} “do ndb = 1, ndbk^{\Sigma}(pe)”\}
      \{\text{generate the tile index } ti_k \leftarrow tile^{\Sigma}(pe, ndb, t_1, \ldots, t_{r-1}, t_{r+1}, \ldots, t_{d_a})\}
      \{\text{generate the statement} \((t_l_k, t_u_k) \leftarrow tile_{\Sigma}(Lb_k)\}\}
      \{\text{generate the statement} \((t_u_k, u_k) \leftarrow tile_{\Sigma}(Ub_k)\}\}
      \{\text{GenLoopBounds(a, } r, ti_k, t_l_k, t_u_k, l_k, u_k; ll_k, lu_k)\}
      \{\text{generate the loop} “do li_k = ll_k, lu_k”\}
      \{\text{generate nb for each array referenced in the body of } i_k\}\}
  endif
  else \(\{c\) is either a conditional or a statement\}
    convert each array reference in \(c\) using GenBlockedReference
  endif
endfor
end

Figure 3.6: Algorithm to parallelize a loop nest with no subroutine calls and no loop-carried dependences.
algorithm GenTilingLoops($i_k$)
given A blockable loop $i_k$ with loop bounds $Lb$ and $Ub$
output code to generate a loop $ti_k$ that steps across
the tiles in $i_k$ and another loop $li_k$ that steps across the points in the tile.

generate the statement $(tl_k, l_k) \leftarrow tile\_elem^a(Lb_k)$
generate the statement $(tu_k, u_k) \leftarrow tile\_elem^a(Ub_k)$
generate the loop “do $ti_k = tl_k, tu_k$”
GenLoopBounds($ti_k, tl_k, tu_k, l_k, u_k; ll_k, lu_k$)
generate the loop “do $li_k = ll_k, lu_k$”
generate the statement “$i_k \leftarrow \sum_{j=1}^{tix} ts_k^a(j) + li_k$”
end

algorithm GenLoopBounds($ti, tl, tu, lb, ub : ll, lu$)
given $ti$ is an index into a tiled dimension $k$ of some array $a$. The set $< tl, tu, lb, ub >$
gives the range of a loop stepping across elements in $a_k$. expressed in terms of
the beginning tile $tl$, ending tile $tu$, the starting element $lb$ in $tl$ and
the ending element $ub$ in $tu$.
output code to generate the bounds $ll, lu$ of an intratile loop

generate the statement $ll \leftarrow 1$
generate the statement $lu \leftarrow ts_k^a(ti)$
generate the statement “if ( $ti = tl$ ) $ll \leftarrow lb$”
generate the statement “if ( $ti = tu$ ) $lu \leftarrow ub$”
end

Figure 3.7: Auxiliary algorithms that aid in parallelizing loop nests.
functions of outer loop variables. Note that the value of the original loop variable is explicitly generated as it may be needed to calculate the loop bounds of an inner loop. Loops in the original loop nest that correspond to untiled dimensions are left unchanged. Each array reference within the nest is converted to refer to these intratile indices. These conversions may require guards, as explained in Section 3.3.4. This set of transformations creates a blocked loop nest that sequentially simulates the actions of each of the processors on each of its local blocks.

If a loop nest \( L \) contains subroutine calls, the loop nests within those subroutines would already have been converted, since the call tree is traversed bottom-up. Since those loop nests already contain a processor loop, \( L \) should not be transformed in the way described above. Each loop in \( L \) which corresponds to a tiled dimension \( k \) and whose body contains a subroutine call is replaced with two loops: the outer one \( ti_k \) stepping across the tiles in \( i \) and the inner one \( li_k \) stepping across the intratile elements. The bounds of the outer loop are \( \{1, \ldots, nt_k\} \) while the bounds of the inner loop are calculated as described above. The code generating the bounds for the inner loop is placed at the beginning of the body of the \( ti_k \) loop before the \( li_k \) loop. Note that the subnests of \( L \) that do not have subroutine calls in their body are still transformed using the algorithm \textit{BasicConvert}. Individual statements that do not belong in call-free subnests are preceded by calculations of \( pe, nb \) pairs for the arrays involved using their respective \textit{procnum} and \textit{peblk} functions. The array references throughout \( L \) are converted as described earlier.

If a loop \( i_k \) in the loop nest \( L \) has a LCD and corresponds to a tiled dimension, it cannot be enclosed by a \textit{pe} loop, since accessing its tiles in the order of the processors that own them may not honor the dependence. However, loops enclosed by \( i_k \) that are
The given algorithm, `LCDConvert`, is designed to parallelize a loop nest with loop-carried dependences (LCDs) but no subroutine calls. It takes as input a loop nest `L` whose loops have loop variables `{i_1, ..., i_n}`, lower bounds `{Lb_1, ..., Lb_n}`, and upper bounds `{Ub_1, ..., Ub_n}`, where each bound may be an integer or a function of outer loop variables. Some of the loops may have loop-carried dependences (LCDs) but none have subroutine calls. The control constructs in `L` are given by the ordered set `S`. Blockable loops in calling routines that may enclose `L` are given by the possibly empty list `{i_1, ..., i_m}`.

The output of the algorithm is a transformed loop nest `L_b` in which loops that correspond to tiled dimensions in any array are blocked to yield new bounds `{lb_1, ..., lb_n}` and `{ub_1, ..., ub_n}`.

The algorithm proceeds as follows:

1. For each perfect subnest `P` of `L` that is free of loops with LCDs, perform the algorithm `BasicConvert(P)`.
2. For each blockable loop `i_k` in `S` with LCDs, perform:
   - Call `GenTilingLoops(i_k)`.
   - If `i_k` does not enclose any blockable loops with LCDs, perform the algorithm `GenShadowCopy` for each use reference involved in an LCD.
3. If `i_k` does not enclose any blockable loops, perform:
   - Let `a` be any tiled array referenced in the body of `i_k`.
   - Let `P` be the set of constructs enclosed by `i_k`.
   - Generate `pe` using `proccnum^a` from the tile indices of the blockable loops.
   - Copy the constructs in `P`, converting array references using the algorithm `GenBlockedReference`.
4. If the above condition is not met, perform `BasicConvert(P)`.
5. Repeat steps 2-4 until all blockable loops have been processed.

Figure 3.8: Algorithm to parallelize a loop with loop-carried dependences but no subroutine calls.
free of LCDs and correspond to tiled dimensions can still be executed in parallel, with a synchronization at the end of each iteration of $i_k$. Also, the results of each iteration of $i_k$ need to be communicated to the processor owning the adjacent tile at the end of the iteration. Note that the interprocessor communication this requires provides the needed synchronization at the end of the $i_k$ loop in the parallel program, while blocked code is sequential and does not need synchronization at all. To reduce the granularity of communication and synchronization, it is desirable that such a loop be at the outermost level possible. This consideration drives the algorithm $LCDConvert$, presented in Figure 3.8, that handles such loops. After processing all LCD-free perfect subnests of $L$, the algorithm interchanges each blockable loop $i_k$ that has a LCD with the outermost loop $i_l$ in its perfect subnest that has no LCD. This is always possible since these two loops cannot be interchanged only if there were a dependence with a direction vector of the form $(\ldots, <, *^{k-l-1}, >, \ldots, *)$ involving $i_l$ and $i_k$ but that would make $i_l$ itself have a LCD, contrary to assumption. This ensures that each loop with a LCD that needs to be blocked encloses the maximum possible number of loops.

A group of blockable loops with LCDs in the same perfect subnest are transformed into a set of loops that step across the tiles in array dimensions, all of which enclose another set of loops that step across elements within those tiles. In each perfect subnest of the original program with the loop interchanges, the innermost blockable loop $i_k$ with a LCD is located. If it encloses no blockable loops, the body of $i_k$ has to be executed sequentially. Otherwise, the code region enclosed by $i_k$ is transformed using the algorithm $BasicConvert$ discussed earlier. Finally, individual statements that are not enclosed by a processor loop are preceded by calculations of $pe.nb$ pairs for the arrays involved using their respective $procnum$ and $peblk$ functions.
Loop nests that contain both subroutine calls and loops with LCDs are treated by a combination of above methods.

**Shadow Copy Generation**

The goal of the methodology is to build an efficient message-passing parallel program. In such programs, message startup overhead is an important factor that affects performance and so it is necessary to minimize the number of individual messages. The main considerations in achieving this objective are avoidance of redundant data transfers, transferring as many array elements as possible in each message and combining the transfer of the shadows of many arrays when possible. The methodology provides a well-defined framework for the user to address these performance issues. Since the shadow copies generated in the blocking pass are mechanically replaced with MPI-based communication in the second pass, these efficiency issues need to be addressed in generating the shadow copies. This section describes the framework that addresses the questions of determining which array references need shadow copies, where the copies are to be placed and the generation of the code that performs the copy.

A use of an array $a$ in the blocked program that accesses a shadow element needs to be preceded by a shadow copy for that element. Determining whether an array reference accesses a shadow element involves determining the range of elements $l_k : u_k$ accessed in each tiled dimension $a_k$. This range is obtained from the bounds of the loops surrounding the reference and so may involve integers or symbolic expressions involving loop variables of outer loops. If $l_k \leq 0$ or $u_k > ts_k(ti_k)$, where $ti_k$ is the index of the tile that the reference uses, a shadow copy is needed for the shadow elements along $a_k$. The set of ranges $l_1 : u_1, \ldots, l_\Delta : u_\Delta$ for a reference to a $\Delta$-dimensional
do k = 1. NumIter
  
  do pe = 0, npe - 1
    do tj = 0, npe - 1
      jl = 0 ; ju = m-1
      if ( tj = 0 ) jl = 1
      if ( tj = m-1 ) ju = m-2
      do lj = jl, ju
        do ndb = 1, 1
          ti = tile^a_\epsilon_(pe.ndb,tj)
          il = 0 ; iu = m-1
          if ( ti = 0 ) il = 1
          if ( ti = m-1 ) iu = m-2
          nb^a = peblk^a_(ti,tj); nb^b = peblk^b_(ti,tj)
          do li = il, iu
            a(li,lj,nb^a,pe) = b(li-1,lj,nb^b,pe) + b(li+1,lj,nb^b,pe) +
            b(li,lj-1,nb^b,pe) + b(li,lj+1,nb^b,pe)
          enddo li
        enddo ndb
      enddo lj
    enddo tj
  enddo pe
enddo k

Figure 3.9: Example: parallelizing a loop without LCDs and subroutine calls.

91
array $a$, which includes ranges of tiled and non-tiled dimensions, is called a *reference descriptor*. The set of elements to be copied is obtained by replacing the ranges corresponding to shadow dimensions in the reference descriptor with the range of the shadow elements only, yielding a *shadow reference descriptor* (SRD). Note that, for a given shadow dimension $k$, the reference descriptor may have to be split into two shadow reference descriptors, one each for the increasing and the decreasing direction of $k$. Since many dimensions in the reference of $a$ may require shadow copies, a single reference descriptor may yield up to $2^d$ SRDs for an array with $d$ tiled dimensions. Each SRD can be used to algorithmically generate a loop nest that copies its shadow elements. This procedure determines the smallest number of elements that need to be transferred.

Shadow copies are needed in two situations. The first situation involves a loop with a LCD in the original program whose equivalent in the blocked program indexes a tiled dimension $a_k$ of an array $a$. The shadow elements of a tile of $a$ need to be copied in each iteration of the loop in the blocked program that steps across the tiles of $a_k$, except possibly the first or the last. The second situation involves a definition of an array $a$ in a loop followed by a use of $a$ with a stencil in another loop. The algorithm for placing the shadow copy treats these two situations differently. In the case of a loop with LCD, the loop nest performing the shadow copy is placed after the loop that steps across the tiles of $a_k$ but before the loop that steps across the elements within the tile. If many loops with LCDs form a part of a perfect subnest, the shadow copy is placed after all the corresponding tile loops but before any of the intra-tile loops. The loop level so chosen is guaranteed by construction not to be
enclosed by a processor loop, so that the loop nest that copies the shadows will need such a loop pair of its own. This is desirable during debugging.

In the second situation, the array use that necessitates the shadow copy is compared with each of its reaching definitions that computed a source element for the needed shadow to decide the maximum common loop level for all of them. The shadow copies are placed at this loop level after the last loop that defines its source elements. (Note that the shadow copy may be placed in this loop level anywhere after the last loop that defines its source elements and before the first loop that contains the array use. The user can exploit this flexibility to merge shadow copies in a given loop level.) It is possible that the maximum common loop level is zero i.e. there may not be a loop that spans the last reaching definition and the use. This placement guarantees that the copy for any given SRD is performed with the largest possible copy size and that there is no redundant transfer of data.

Since it is desirable to place shadow copies in a processor loop not involving computation, if the loop level chosen for the shadow copy is enclosed by a processor loop, the loop is split in two so that the reaching definitions are confined to the body of the first loop and the use falls in the second. This loop fission is always legal because the methodology never places such a processor loop around loops with LCDs so that the body of the loop can be split safely.

The actual generation of the copy is done using the algorithm *GenShadowCopy*, presented in Figure 3.10. A processor loop is created and tile indices of the dimensions in the array $a$ whose corresponding loops do not enclose this loop level are calculated. Since the copies will simulate a data pull from another processor, the calculated tile indices refer to the sink tile whose shadows need to be filled. As discussed earlier, there
algorithm GenShadowCopy

given • a shadow reference descriptor \( lb_1:ub_1, \ldots, lb_\Delta:ub_\Delta \) of a use reference \( r_a \) of an array \( a \)
• the set of shadow dimensions in \( r_a \)
• loop level \( \lambda \) where the shadow copy is to be placed
• set \( L \) of loops enclosing \( r_a \) at loop levels greater than \( \lambda \)
• \( d \), the number of tiled dimensions in \( a \)

output a loop nest that copies the shadows in the given SRD.

generate a loop “do \( pe = 1, npe \)”  
foreach non-shadow dimension \( a_k \) in \( r_a \) indexed by a loop \( i_k \) in \( L \) do
    if \( a_k \) is a tiled dimension then
        GenTilingLoops(\( i_k \))
        \{Let the loop variable of the tile loop be \( t_{i_k} \).\}
        compute tile index in \( a_k \) of the source tile as \( ti_k^S \leftarrow t_{i_k} \)
    else
        generate the loop “do \( i_k = lb_k, ub_k \)”
    endif
endfor

foreach shadow dimension \( a_k \) in \( r_a \) do
    foreach shadow dimension \( a_l \neq a_k \) do
        \{Let \( i_l \) be the loop that indexes \( a_l \).\}
        GenTilingLoops(\( i_l \))
        \{Let the loop variable of the tile loop be \( t_{i_l} \).\}
        compute tile index in \( a_l \) of the source tile as \( ti_l^S \leftarrow t_{i_l} \)
    endfor

    generate the loop “do \( ndblk = 1, nblk^a(\text{pe}) \)”
    generate the tile index for \( a_k \) as “\( ti_k = \text{tile}(\text{pe}.\text{ndblk}. \text{ti}_1, \ldots, \text{ti}_{k-1}, \text{ti}_{k-1}, \ldots, \text{ti}_d) \)”
    generate the source tile index in \( a_k \) as \( ti_k^S = ti_k - 1 \) if \( lb_k \leq ub_k \leq 0 \).
    \( ti_k^S = ti_k + 1 \) otherwise

    generate the source processor as “\( \text{pe}^S \leftarrow \text{procnum}(\text{ti}_1^S, \ldots, \text{ti}_d^S) \)” and
    the source block as “\( \text{nb}^S \leftarrow \text{peblk}(\text{ti}_1^S, \ldots, \text{ti}_d^S) \)”

    generate the loop “do \( s = lb_k, ub_k \)”
    generate the source point as \( s^S = ts_k^a(ti_k^S) + s \) for \( s \leq 0 \) and
    \( s^S = s - ts_k^a(ti_k^S) \) for \( s > ts_k^a(ti_k^S) \)
    generate “\( a(i_1, \ldots, i_{k-1}, s, i_{k+1}, \ldots, i_\Delta, \text{nb}, \text{pe}) = \)
    \( a(i_1, \ldots, i_{k-1}, s^S, i_{k+1}, \ldots, i_\Delta, \text{nb}^S, \text{pe}^S) \)”
endfor
end

Figure 3.10: Algorithm to generate code that copies the shadow elements of an array.
may be loops enclosing the processor loop that step across tiles in a or a colocated
dimension in another array. So, the shadow copy must be performed only when
the tile indices match the loop variables of those enclosing loops. This is done by
generating guards suitably. Also, the extreme tiles $ti = 1$ and $ti = nt_d$ of a shadow
dimension $a_d$ may not need shadow copies: this is also ensured by generating guards
that check for extreme values. Each index $ti^S_k$ of the source tile of the shadows is
then calculated from the corresponding sink tile index $ti^T_k$. The set of source tile
indices enables the generation of code that calculates the source processor index by
using the procnum and peblk functions. Blocked loops are first generated for non-
shadow dimensions which do not have a corresponding loop at this loop level. For
each shadow dimension, a blocked loop is generated for every other shadow dimension.
This ensures that ‘diagonal’ shadow elements (i.e. those whose source tiles differ in
more than one dimension from the sink tile) are properly handled.

The user may use this basic framework to tune performance by merging shadow
copy loop nests at the same loop level that may correspond to different shadow
reference descriptors of either the same array or different arrays. Such loop fusion
can be eased by ranking the dimensions of all arrays in the program globally, with
colocated dimensions sharing the same rank, so that the blocked loops in algorithm
GenShadowCopy can be generated in order of rank. If a loop of lower rank has its
bounds calculated from an outer loop of higher rank, conflicts will arise, which the
user needs to resolve.
Figure 3.11: Example: Shadow copy.
The MPI Pass

The blocking pass yields a sequential program with both the original code and the blocked code that produces the same output as the original. This needs to be converted into a parallel program in which each processor accesses only its local data, the shadow copies are replaced by communication across processors. These communications also serve to synchronize processors that handle the iterations of loops with LCDs. This is accomplished by the MPI pass in three stages, again in an incremental manner. These stages are discussed further in this section.

The first stage of the MPI pass begins by building a parallel program that essentially executes the sequential version built by the blocking pass on many processors. This is done by adding a call to the MPI initialization routine MPI_Init() at the beginning of the program and a call to the MPI termination routine MPI_Finalize at the end. Thus, all computations are completely replicated among the processors and the shadow copies are still retained. Then, each of the shadow copies is supplemented with a MPI communication. That is, following a shadow copy from source processor $pe'$ to sink processor $pe$, a communication is placed in which the processor with $myid = pe'$ sends a set of array elements and the processor with $myid = pe$ receives it and updates its shadow elements. The algorithm by which a loop nest performing a shadow copy is transformed into a loop nest performing MPI communication is presented in Figure 3.12. However, the shadow copies are performed for all values of the $pe$ index while the communication-based shadow update is performed only for $pe = myid$. This is so because each processor needs to carry out the blocked routine computations for all values of $pe$ until all interprocessor communication is in place.

Another point of note is that the MPI communication is placed after the shadow copy
algorithm GenMPI

given • a shadow copy loop nest $L$ that has a single statement of the form
$$a(i_1, \ldots, i_{d-1}, s, i_{d+1}, \ldots, i_\Delta, nb, pe) = a(i_1, \ldots, i_{d-1}, s^S, i_{d+1}, \ldots, i_\Delta, nb^S, pe^S)$$
• two buffers $\textit{sendbuf}$ and $\textit{recvbuf}$ of size $r_1 \ast r_2 \ast \ldots \ast r_{d-1} \ast r_{d+1} \ast \ldots \ast r_\Delta$. where the $r_k$ are the ranges of the shadow reference descriptor of the reference that necessitates the shadow copy.

output a loop nest that fills the shadows specified by the SRD using MPI communication

generate the message size msg.sz as $r_1 \ast r_2 \ast \ldots \ast r_{d-1} \ast r_{d+1} \ast \ldots \ast r_\Delta$
generate the $pe$ loop and the associated computations of tile indices, source processor id and the guards
generate a guard “if (myid = $pe^S$)”
copy the loops of $L$ except for the $pe$ loop
generate the current index indx of $\textit{sendbuf}$
generate the statement “$\textit{sendbuf}(\text{indx}) = a(i_1, \ldots, i_{d-1}, s, i_{d+1}, \ldots, i_\Delta, nb, pe)$”
terminate the loops of $L$
generate a call to MPI\_SEND to communicate $\textit{sendbuf}$ of size msg.sz to $pe$
terminate the guard

generate a guard “if (myid = $pe$)”
copy the loops of $L$ except for the $pe$ loop
generate the current index indx of $\textit{recvbuf}$
generate the statement “$a(i_1, \ldots, i_{d-1}, s, i_{d+1}, \ldots, i_\Delta, nb, pe) = \textit{recvbuf}(\text{indx})$”
terminate the loops of $L$
generate a call to MPI\_RECEIVE to receive $\textit{recvbuf}$ of size msg.sz from $pe^S$
terminate the guard

terminate the $pe$ loop

end

Figure 3.12: Algorithm to transform a loop nest with a single shadow copy statement into a loop nest with equivalent MPI.
so that it overwrites some of the shadow elements that were updated by the shadow copy. This ensures that errors in the communication will affect the subsequent computations and thus manifest as differences reported by the compare routines. When all interprocessor communication is in place, the first stage is complete.

The second stage removes the replication of computations. This is done by conditionally executing the body of each pe loop except those involved in MPI communication only for pe = myid. This renders all shadow copies invalid because the source elements of the shadow copy will not be computed correctly. So, the shadow copies are suppressed. This results in a parallel program with purely local computations that uses interprocessor communication to update the shadow elements. However, it still retains the original sequential code and the compare routines.

The third stage eliminates the processor dimension of all the arrays since each processor accesses only one element in it. This can be done by a global replacement of the last subscript of all array references and the last dimension of all array declarations. This stage also eliminates all the sequential code and the compare routines, resulting in a purely parallel program.

### 3.4 Performance Measurement

The methodology is designed to simplify the generation of parallel programs without compromising performance. The performance of the program *fdl3di* in the case study was measured to verify that the design goals are met. The results of the performance study are reported and discussed in this section.

The performance of *fdl3di* was investigated in four modern parallel computing systems: the IBM SP2, the Silicon Graphics Power Challenge, the Cray T3E and
the Silicon Graphics Origin 2000 1. The IBM SP2 is a scalable distributed memory system with 256 processing elements, expandable to 512 nodes, of which 230 are used as compute nodes. Each node is a 135 MHz Power2 processor, with on-chip L1 caches for data (128 KB) and instructions (32 KB), 1 GB of main memory, and a peak performance of 500 Mflops. The Power Challenge is a bus-based SMP system with 16 nodes, expandable up to 36 nodes, with 2048 MB of main memory. Each node is a 90 MHz IP21, containing a MIPS R8000 microprocessor (Chip Revision 3.0), augmented with a R8010 FPU (Chip Revision 0.2). The primary on-chip data and instruction caches are 16 KB in size, while the secondary unified cache is 4 MB. The Cray T3E is a distributed memory scalable system, with 128 processing elements, expandable to 512. Each processing element is a 300 MHz DECchip 21164, with on-chip direct-mapped L1 caches of 8 KB each for instruction and data, and a unified on-chip three-way set-associative L2 cache of 96 KB. Each processor has access to 128 MB of local memory and can deliver up to 600 Mflops. All run a 64-bit operating system, AIX 1.4 in the case of the SP2, UNICOS/mk on the T3E and IRIX 6.2 in the Power Challenge.

For parallel programs, elapsed wall clock time is a good measure of performance. The main loop of fd13di was timed using calls to the gettimeofday() system call in UNIX, which can be used to measure wall clock time to a precision of a few microseconds. The elapsed wall clock times are reported by each MPI process and the largest of these is taken as the execution time of the program. To compensate for the impact of system load on program execution times, each version was run four times and the

1 The IBM SP2 and the Origin 2000 were accessed from the Information Technology Laboratory at the U.S. Army Corps of Engineers Waterways Experiment Station (CEWES), while the Power Challenge and the T3E are located at the Ohio Supercomputer Center.
lowest of the four execution times was taken as the final measure of performance. The program was run with a problem of size 100, i.e., the grid resolution was set at $100 \times 100 \times 100$ and the main loop was run for 3 iterations, each with 2 iterations of the inner loop, in all the runs. On the IBM SP2 and the Cray T3E, the programs were compiled with the highest level of non-aggressive compiler optimization, -O2. However, on both the Silicon Graphics machines, this setting causes the output of the parallel program to differ from that of the sequential counterpart: hence the compiler flag used was -O1. Hence the absolute execution times on these different platforms are not comparable.

Figure 3.13: Execution times, in seconds, for a $100 \times 100 \times 100$ problem on various parallel computers.
The execution times for various processor counts on different parallel computing platforms are presented in Figure 3.13 and the corresponding speedups are presented in Figure 3.14. For the SP2, it is notable that the 2-processor run executes more than twice as fast as the single processor run, resulting in a superunitary speedup. This is because the amount of data per processor decreases as the processor count increases so that there is better data locality in the cache. The same factor accounts for the superunitary speedup observed with 4 and 8 processors. However, when the processor count reaches a threshold at which the needed data fits in the cache, this trend does not continue. Thus, for a processor count of 16, the speedup is not superunitary but still significant.
The performance figures for the Power Challenge show substantial speedup until 8 processors and, for 16 processors, it is almost 9. However, since the Power Challenge has a much larger cache than the SP2, the superunitary behavior is not observed.

On the Cray T3E, a problem of size $100 \times 100 \times 100$ is too large to fit within the main memory available on a single processor. Since the system does not support virtual memory, the least number of processors on which the problem could be run is 4. Hence data is presented for processor counts ranging from 4 to 24. The program is seen to exhibit a speedup, relative to the 4-processor execution time, of 1.8 on 8 processors and 2.90 on 16 processors. As the processor count increases further, the speedup drops due to the serial component of the implementation.

The Origin 2000 has some of the largest absolute execution times but it exhibits almost linear speedup until 8 processors and sustains good performance until 24 processors.

3.5 Related Work

The problem of automatic conversion of sequential programs to a concurrent form has received considerable attention. The main tasks involved in such a conversion are the determination of the best data partition that maximizes load balance and minimizes communication, the analysis of the loops in the sequential program to determine which can be parallelized or transformed to a form that can be parallelized, and the generation and placement of communication and synchronization primitives to ensure correct fast execution of the generated code.

There have been several surveys of tools and projects that attempt an automatic or semi-automatic parallelization of sequential numerical codes [11, 13, 30]. These
approaches vary in the level of generality at which they attempt to solve the problem. While some attempt to perform all the tasks mentioned above, others address only the issues related to detecting concurrency and code generation. Also, some tools have aimed only at specific classes of applications and/or allow only some data partitioning schemes. This section gives a brief description of the various approaches and compares and contrasts them with this thesis.

Past research projects that have attempted automation of all the steps involved [4] encountered a fundamental problem: choosing the optimal data partitioning scheme is a NP-complete graph theoretic problem [18] problem and hence difficult to solve in the general case. Thus, although these approaches synthesize programs with good performance for regular computations with blocked or cyclic data distributions [38], far more research has to be done to make them viable in the general case.

Another set of studies have addressed this difficulty by accepting directives from the user for determining the data partitioning scheme. This is the approach taken by High Performance FORTRAN [14, 27], the proposed standard for writing distributed memory programs, and also FORTRAN D [46]. These parallel programming environments allow the programmer to augment a given sequential program with directives that specify the alignment of arrays with respect to programmer-defined templates and also the partitioning scheme which allocates tiles of the templates to the processors. Programmers may also specify using compiler directives that specific loops are parallelizable. These features have enabled the synthesis of code with good performance for a large class of applications. However, these tools aim at the conversion of any general sequential program; this introduces overheads in the converted program
that reduce performance for many applications. Furthermore, in the current state-of-the-art, they handle only simple data distributions, such as blocked and cyclic distributions. These factors limit the performance of the synthesized code. For example, Hayder et al. [22] compared the performance of an application that solves a partial differential equation on a regular grid in two cases: one where the application uses the PETsc library of parallel routines, which is written using MPI, and another where the application is synthesized using HPF. The results show that the PETsc implementation exhibits higher speedups and lower execution times than the HPF-synthesized version. Similarly, a study by Roe et al. [33] comparing a numerical application hand-coded using MPI with the corresponding HPF programs generated by various available HPF compilers points out the need for further improvements in compiler technology for HPF to be useful in a broad range of applications.

Another way to allow users to choose the data partitioning scheme is to develop interactive tools that work with a parallelizing compiler and aid the user in the decision-making process. This approach is typified by the Interactive D-Editor [23] and Forge [17]. However, they also have limitations in dealing with legacy codes that may need complex data distributions [31, pp.2].

Other researchers have attempted to overcome the performance limitations arising from overheads by focusing on specific problem domains so that domain-specific information can be used in optimizing the synthesized code. For example, the Parallelization Agent [31, 26] focuses on regular grid-based computations that typically arise in domains such as environmental modeling. It uses a knowledge-based approach, similar to expert systems, to handle the large search spaces that arise in the subproblems described earlier.
The approach proposed in this thesis avoids the problems associated with automated choice of data distributions by allowing the user to specify the data distribution scheme. However, it differs from the HPF approach as it allows any tiled distribution, including skewed ones, whereas the HPF standard allows only block and cyclic distributions. Also, since it focuses on a specific class of applications, namely stencil computation, it eliminates the overheads associated with general purpose tools. More importantly, it differs from both HPF and other domain-based approaches by allowing the programmer to alter the generated program manually to improve performance. This can be accomplished by altering the blocked sequential form of the program to optimize communication or computation in an incremental way and validating the changes before resynthesizing the final parallel program. We are not aware of any HPF-based or domain-specific tool that provides any such assistance for controlled modification of the synthesized parallel program.

3.6 Conclusion

This chapter has presented a methodology of parallelizing a large class of sequential programs that is based in the philosophy of targeting a well-defined class of applications and allowing considerable flexibility to the user while still assisting her with providing programmatic assistance and systematic ways to track conversion errors. Such an approach enables relatively rapid parallelization of stencil codes without sacrificing performance. The essence of the methodology is a two-pass approach, in which the first pass chooses a tiled data distribution and mapping for each array and incrementally transforms each routine into a blocked form that simulates the action of many processors on these tiled arrays. The second pass incrementally augments
each shadow copy in the blocked program with an MPI-based communication before suppressing the shadow copies to derive a purely parallel program.

Performance of the resulting program was evaluated on many commercially available multicomputers. It was demonstrated that the MPI-based program generated by the methodology exhibits good speedup for a range of processor counts.
CHAPTER 4

ALTERNATIVES FOR PARALLELIZATION IN SHARED ADDRESS SPACE COMPUTERS

4.1 Introduction

On computing systems that offer a single address space model and a compiler that performs automatic parallelization, it is relatively easy to convert a sequential program to a parallel form that operates in a shared address space. However, automatically parallelized programs may perform poorly due to lack of data locality, as discussed further below. To ensure data locality, the global data space of the sequential program must be explicitly distributed among the processors. This is naturally achieved when a sequential program is rewritten for execution in a distributed address space model, such as the model provided by MPI, as the global data space needs to be partitioned among the processors.

The methodology proposed in Chapter 3 allows the development of MPI-based parallel programs for a large category of stencil applications. While the methodology could be automated, tools that implement it are yet to be developed so that it must be carried out by hand at present. Compared to traditional methods of manual parallelization, which frequently require that the sequential program be rewritten from
scratch in a parallel form, this needs far less effort and eases multimember conversion projects by allowing team members to work relatively independently. However, the time and effort needed for such a conversion is still much greater than those needed for automatic parallelization.

This study proposes another alternative, based on the methodology, for cache-based shared address space multiprocessors, that exploits data locality to reduce finish time while taking less effort to develop than an MPI-based program. The key observation that makes such an alternative possible is that the blocked sequential program that is produced from the original sequential program as part of the methodology distributes its data among a set of \( n \) virtual processors such that off-processor data accesses are minimized. The blocked sequential program can be converted to execute as \( n \) parallel threads, each of which accesses data in a local manner while interthread communication is implemented as copying of shadow values within the shared address space that the threads execute in. The thread-based parallel program can be developed from the blocked sequential program with less effort than an MPI program because, while the development is still incremental, there is no need to add extra code to the program and the development proceeds in a single address space, which is easier to debug. Furthermore, such an approach allows the user to avoid much of the details associated with interprocessor communication and synchronization, as discussed below (see Section 4.3). Thus, such a thread-based program, called the blocked parallel program provides an alternative that offers potentially better performance than automatic parallelization while requiring less effort than conversion to MPI form.
The case study involving fdl3di, an application in computational fluid dynamics, which was discussed in Chapter 3, was extended to compare the performance of the parallel programs produced by the three alternatives described above: automatic parallelization, conversion to the distributed address space of MPI and the conversion to blocked parallel form. The performance evaluation was conducted on both a symmetric multiprocessor (SMP) architecture with caches as well as a cache-coherent non-uniform memory access (ccNUMA) architecture. The results of the evaluation, presented below in Section 4.4, confirm that the blocked parallel program exhibits performance comparable to the MPI-based program but much better than the autotoparallel program.

The rest of the chapter is organized as follows. Section 4.2 shows that the lack of data locality in autotoparallel programs can degrade their performance. Section 4.3 proposes an extension to the methodology to build a blocked parallel program from the blocked sequential program. Section 4.4 presents the results of the performance analysis in the case study and analyzes the performance measures. Finally, Section 2.10 summarizes the chapter and highlights possible future developments that could impact the alternatives discussed above.

4.2 Automatic Parallelization: Pros and Cons

Autotoparallelizing compilers for shared memory multiprocessors usually generate programs that exploit loop-level parallelism. They may accept a plain sequential program and analyze it to detect the parallel loops or they may accept an annotated program in which certain loops are marked as parallel through compiler directives. Even if the program needs to be annotated, that takes far less effort and time compared to
manual conversion to a distributed address space form, even when the methodology is employed. On the other hand, the parallel programs generated by such compilers tend to perform poorly. To see the underlying reason for the poor performance, consider the execution of the program fragment below on a cache-based multiprocessor.

```fortran
real a(imax, jmax)
do i = 1, imax
  do j = 1, jmax
    a(i,j) = ...
  enddo j
enddo i
```

```fortran
do j = 1, jmax
  a(i,j) = ...
enddo j
```

The program fragment has two loop nests, the first of which computes a two-dimensional array `a(imax, jmax)`, with the second using and defining it. If both the loop nests were parallelized, a portion of `a` which is handled by a processor `p` in the first loop nest could be handled by another processor `q` in the subsequent loop nest. This would cause that array portion to migrate from the cache of processor `p` to that of `q`. On a symmetric multiprocessor (SMP), this would require that, for each element in the array portion, the corresponding entry in the cache of `p` be invalidated, that the main
memory be updated (if a write-back policy were followed) and that the element be placed in the cache of \( q \) before processor \( q \) could operate on that element. Similarly, on a ccNUMA system, this would cause the cache entry of \( p \) to be invalidated, the main memory to be updated (if a write-back policy were followed) and the relevant page in main memory to be transmitted over the network to \( q \) before processor \( q \) can access it. The latency of these operations can drastically increase the execution time of the second loop nest, especially if the processor interconnection network is congested. So, for applications that are composed of several parallelizable loop nests, performance may suffer.

Such a scenario can be avoided if an explicit data distribution were specified for the array \( a \) and the compiler takes it into account when mapping iterations to processors. However, the proposed standard for autoparallelization for shared memory multiprocessors, the OpenMP model [16], does not provide for directives that specify data distributions. Furthermore, the compilers that support such a specification allow only a restricted class of data distributions such as blocked or cyclic distributions. As was shown in Chapter 3, some applications such as \( fdl3di \) perform best with other data distributions such as a skewed block one.

Thus, automatic parallelization is not suitable for all applications, even though it offers a fast approach to parallelization. In particular, for the case study, the application \( fdl3di \) performs best with a skewed block distribution and hence cannot be expected to do well under automatic parallelization. This is confirmed by the performance evaluation presented in Section 4.4.
4.3 Generating the Blocked Parallel Program

Conversion to a blocked parallel program is an alternative to automatic parallelization for a large class of stencil computations that come under the scope of the methodology. The development of the blocked parallel program is easier and takes less time compared to the conversion to an MPI-based program, which is the other alternative available to parallelize such computations. This section describes the process by which the blocked parallel program is generated.

The blocked parallel program is a thread-level concurrent program and is built by incremental conversion of the blocked sequential program. The conversion process is similar to the generation of the MPI program from the blocked sequential program in the MPI Pass of the methodology (see Section 3.3.6). The conversion of a subroutine is accomplished by enclosing each of its loop nests with compiler directives that declare the outermost parallelizable loop in the nest as dependence-free. The methodology ensures that, for loop nests without loop-carried dependencies, the outermost such loop is always the processor loop. Then, the computation in the loop nests and the shadow copying operations are 'localized', in exactly the same manner as in the MPI Pass. This transformation allows a region of the code to execute in parallel while the unconverted regions still execute in sequential mode. Note that the called subroutines need to be converted before converting the calling subroutines so that the program's call tree needs to be traversed bottom-up.

In an MPI program, processors communicate with explicit exchange of messages. This ensures that the sending process and the receiving process are synchronized, i.e., the receiver never receives invalid data. However, in a blocked parallel program, the threads communicate among each other through shadow copies. This does not
automatically ensure synchronization of the sender and the receiver. It is necessary for the programmer to ensure that, when a processor copies shadow values from the array segments owned by another processor, the concerned segments contain valid data. In other words, the owning processor needs to have updated those segments before the copying takes place. This calls for synchronization among the processors. While point-to-point synchronization among processors may be employed to ensure the correctness of copies, the details involved in such low level coding make the process error-prone and tedious. Indeed, much of the difficulty associated with the MPI Pass arise from the need to handle these details of processor indices and buffer addresses. Synthesis of the blocked parallel program can circumvent these issues by using barrier synchronizations in the place of point-to-point synchronizations.

Such barrier synchronization calls need to be placed before each loop nest that performs a shadow copy to ensure that the receiver copies the shadow values only after they have been computed. However, if a loop copies shadow values of an array and also updates that array, it may be necessary to place a barrier synchronization call after the shadow copy also, to ensure that the owner does not modify the values to be communicated before the receiving processor copies it.

While this approach may incur more overhead than the point-to-point synchronization used by MPI programs, it may be expected that this approach both improves the parallelization process by avoiding error-prone tasks and achieves most of the performance gains of conversion to MPI by paying attention to data partitioning issues. This expectation is borne out by the performance study presented in the next section.
4.4 Performance Evaluation of *fdl3di*

The case study presented in Chapter 3, relating to *fdl3di*, an application in computational fluid dynamics, was extended to study the relative performance of the three methods of parallelization: automatic parallelization, conversion to an MPI-based form and conversion to a blocked parallel form. This section describes the approach adopted in the performance evaluation and analyzes the data collected to demonstrate that the blocked parallel program performs as well as the MPI-based program but much better than the autoparallel program, although it takes less time to be developed than the MPI-based program.

The time needed for conversion from the blocked sequential form of *fdl3di* to the MPI-based form was several months whereas the corresponding conversion to blocked parallel form took less than a week. This experience confirms the idea that conversion to blocked parallel form would be faster.

The performance of the autoparallel program, the blocked parallel program and the MPI program were compared on two parallel computing systems, the Silicon Graphics Origin 2000 and the Silicon Graphics Power Challenge. The former is a ccNUMA system with 16 nodes, expandable up to 128 nodes, with 12.288 MB of main memory. Each node is a 195 MHz IP27, composed of a MIPS R10000 microprocessor (Chip Revision 2.6), assisted by a R10010 FPU (Chip Revision 0.0). The system provides on-chip L1 data and instruction caches, both of size 32 KB, and an unified L2 cache of size 4 MB. The Power Challenge is a bus-based SMP system with 16 nodes, expandable up to 36 nodes, with 2048 MB of main memory. Each node is a 90

---

2 The Origin 2000 used for the study is located at the Information Technology Laboratory at the U.S. Army Corps of Engineers Waterways Experiment Station (CEWES) while the Power Challenge was accessed from the Ohio Supercomputer Center.
MHz IP21, containing a MIPS R8000 microprocessor (Chip Revision 3.0), augmented with a R8010 FPU (Chip Revision 0.2). The primary on-chip data and instruction caches are 16 KB in size, while the secondary unified cache is 4 MB. Both run a 64-bit operating system, IRIX 6.4 in the case of the Origin 2000 and IRIX 6.2 in the Power Challenge.

Both the systems provide Release 3.0 of the implementation of MPI by Silicon Graphics. This is a 64-bit implementation of Version 1.2 of the MPI standard specified by the MPI Forum [15]. This implementation classifies all messages into three categories: short (less than or equal to 64 bytes in length), medium (between 65 and 16383 bytes inclusive) and long (16384 bytes or above). Short messages are always buffered in the shared memory and long ones never buffered but medium messages may or may not be buffered, depending on the number of buffers allocated for this purpose by the system and on a user-defined buffer threshold. The default value for the threshold in this implementation is 16384 bytes, which implies all medium messages are buffered. All the messages in the experimental runs of the MPI-based fdl3di were of medium length. The threshold was not altered so that all messages were buffered. Furthermore, the size of each such buffer, 16 KB, was expected to be large enough to accommodate each message and hence was not altered. Each process typically sends one message at a time and so the number of buffers per process was also not altered.

Both systems provide two facilities for automatic parallelization. The first is the use of compiler directives for declaring loops as parallel. There are two such sets of directives available on each system. The first set, is primarily composed of one directive, the $doacross$, which specifies the loop that immediately follows it as
parallel. This incurs little overhead and hence was used in the autoparallel program as well as the blocked parallel program. The second set, based on the recommendations of the Parallel Computing Forum (PCF), provides the abstraction of a parallel region, in which a pair of directives enclose a region of code, thus specifying the region to be concurrent. Within a parallel region, loops may be marked as concurrent and various kinds of worksharing constructs – such as critical sections of code – may be specified. However, the entry and exit to each parallel region as well as the worksharing constructs within parallel regions require a synchronization of all threads executing the program. These were used to develop the blocked parallel program due to their ease of use. But the blocked program was eventually converted to use the c$doacross directive, to make its performance measures compatible with those of the autoparallel program.

The other facility is the Power FORTRAN Accelerator (PFA), a compiler that can be used to automatically analyze loops for concurrency. The PFA detects that the two outer loops in spenta, the module in fdl3di that solves a pentadiagonal system of linear equations, are sequential but the innermost loop is parallelizable. It parallelizes the innermost loop but this leads to worse performance than if the loop were not parallelized. Hence, the use of the PFA was discarded in favor of using explicit directives.

The blocked parallel program used in this study is a penultimate version in which the principal loop of spenta is executed sequentially, so that the entire equation solver has no parallelism. While the final version achieves parallel execution of the principal loop, it was not ready for performance measurements at the time of the study. while

117
the reported numbers reflect the general trend that may be expected with the final version. the latter can be expected to perform better.

For parallel programs, elapsed wall clock time is a good measure of performance. The main loop of *fdl3di* was timed using calls to the *gettimeofday()* system call in UNIX, which can be used to measure wall clock time to a precision of a few microseconds. For the MPI program, the elapsed wall clock times are reported by each process and the largest of these is taken as the execution time of the program. For the blocked parallel and the autoparallel programs, only the master thread reports the elapsed wall clock time, which is identical with the execution time. To compensate for the impact of system load on program execution times, each version was run twice and the lower of the two execution times was taken as the final measure of performance. All the concerned programs were compiled with the optimization flag of *-O1*.

The three versions of *fdl3di* were run with a problem size of 100, i.e., the grid resolution was set at 100 × 100 × 100. The execution time needed by this problem size exceeded the process time limits on the installation of the Power Challenge; hence a size of 60 was used there. The processor count was varied from 1 to 16 for the study. However, the blocked parallel version, when run with one thread, violated the system's per-process time limits on the Power Challenge; hence, its performance was studied with two or more threads on that system.

The execution times in seconds for the three programs on the Origin 2000 are presented in Figure 4.1. The performance of the autoparallel code improves as processor count increases to 4 but declines thereafter. This is due to the decreasing data locality, as discussed earlier. The MPI program suffers from high overhead and the message startup costs; hence for relatively small number of processors, its absolute execution
Figure 4.1: Execution times in seconds for a problem of size $100 \times 100 \times 100$ on the Silicon Graphics Origin 2000, for the autoparallel, blocked parallel and the MPI programs.
time is worse than the autoparallel version. However, as the processor count increases to 4 and beyond, the superior data locality characteristics of the MPI program cause it to perform better. Also, the performance of the MPI program improves consistently with processor count for the range studied. Since the blocked parallel program suffers from higher overhead and also the cost of barriers, its execution time is the highest among the three for relatively low processor counts. However, as with the MPI program, the reduction in off-processor data accesses resulting from the careful choice of data partition enables it to perform better than the autoparallel version for 8 or more processors.

Figure 4.2: Execution times in seconds for a problem of size $60 \times 60 \times 60$ on the Silicon Graphics Power Challenge, for the autoparallel, blocked parallel and the MPI programs.
On the Power Challenge, the performance results presented in Figure 4.2 show that the autoparallel code shows marginal speedup for low processor counts, attaining a maximum speedup of 2.96 on 8 processors. Its performance worsens for higher processor counts. The MPI program has a higher execution time on one processor than the autoparallel case but performs better for 4 processors and beyond. The blocked parallel code can be seen to speed up consistently with processor count and performs better than the autoparallel code for 16 processors. Thus, the major trends observed in the Origin 2000 are also seen in the Power Challenge.

Note that the blocked parallel version implements synchronization before shadow copies by invoking barriers. The synchronization overheads can be reduced by replacing barrier synchronizations with point-to-point synchronization. However, the good performance of the blocked parallel program even in the absence of this optimization suggests that appropriate choice of data partitioning is the primary factor that affects the performance of parallelized stencil computations.

4.5 Conclusion

This chapter has contrasted the approach of automatic parallelization on shared memory multiprocessors, which can yield poor performance on many applications due to lack of data locality, with the approach of conversion to a MPI-based parallel program, which exploits data locality well to improve performance but is relatively more time-consuming and difficult. A new alternative, the blocked parallel program, was proposed for a large class of stencil computations based on the methodology developed in this thesis, which offers the potential of good performance comparable to the MPI program but takes less effort and time to develop. A performance study
was conducted on two classes of shared memory architectures, namely ccNUMA and SMP, which demonstrated that the blocked parallel program realizes its potential.
CHAPTER 5

CONTRIBUTIONS

This dissertation has underlined the importance of data locality in parallel system performance. Data locality has been studied in two contexts: that of collective communication in existing applications to minimize off-processor accesses and that of conversion of sequential programs to a parallel form that achieves high performance by appropriate data distribution.

Collective communication patterns frequently arise in many applications. Hence it is important to reduce their time to completion. An important global communication pattern, the all-to-all personalized (complete) exchange, was studied in this dissertation in the context of execution in a 2D wormhole-routed mesh. A new algorithm called Cyclic Exchange was proposed that combines the transfer of data from a processor to a set of other processors in a different quadrant of the mesh, thereby reducing the number of messages transmitted over the processor interconnect and hence the completion time. This algorithm was shown to be faster than Quadrant Exchange, a previously proposed algorithm that also adopts a combining strategy [41]. Furthermore, the class of combining algorithms for complete exchange was shown to have benefits complementary to those of algorithms that transfer data directly between processors without combining. This motivated an approach that
hybridizes a combining algorithm. Cyclic Exchange, with a direct one, a member of Scott's family of algorithms, to realize the benefits of both. The behavior of the two algorithms as well as the family of hybrid algorithms were modeled analytically, taking into account the contention among messages for mesh links. A systematic performance evaluation was undertaken to characterize the performance of the hybrid algorithms, using analytical modeling, detailed simulation as well as implementation on commercially available parallel computers with a grid-based interconnect. The performance analysis demonstrated that the hybrid approach performs better than either pure algorithm for a large range of system parameters and problem sizes. The degree of message contention for mesh links was quantitatively measured and shown to have a large impact on performance. The use of barrier synchronization to control and reduce contention was studied as part of the performance analysis and was shown to be effective.

The second part of the dissertation investigated the importance of exploiting data locality in the conversion of numerical programs to a parallel form. The need for a methodology that eases manual parallelization was stressed and such a methodology was developed. The methodology eases conversion by adopting an incremental subroutine-by-subroutine approach, by decoupling data partitioning issues from communication issues and by operating in a single address space model for much of the development. The methodology was developed in the context of a case study involving a specific application in fluid dynamics but its main principles and code transformations were generalized so as to be applicable for a large class of stencil applications. These code transformations were presented in an algorithmic form that allows the methodology to be automated. Performance of the resulting program was analyzed.
and found to be good. On shared address space computers, the methodology makes possible a new alternative that takes less time to develop than MPI-based programs and yet performs as well as them.
APPENDIX A

DATA MOVEMENT IN CYCLIC EXCHANGE

There are issues of organizing data and copying data blocks that arise in any combining algorithm. The time to copy data blocks is an overhead that reduces performance. This appendix presents the details of the scheme used to overlap most of this data movement with communication.

For the sake of clarity, the basic version of Cyclic Exchange, described in section 2.3, is now called Level 1 Cyclic Exchange.

Data need to be organized appropriately so that, in each communication, the set of data blocks to be sent are contiguous in the data array. This necessitates two kinds of movement of data blocks: reorder between phases and data movement within each phase. In each phase, a processor receives data of which some is meant for it and some for other processors. So, between phase $i$ and $i-1$, the processor needs to gather the various data blocks needed for communication in phase $i-1$. This process is called reordering and is typical of combining algorithms.

The set of blocks within each node is organized as a linear array. In phase $i$ of the algorithm, the four quadrants of the data array each contain data blocks for processors in the four quadrants of size $2^i \times 2^i$. If every processor organizes its data quadrants in a fixed sequence, some processors will be forced to communicate two quadrants that
Figure A.1: Movement of Data by Processors in the Four Quadrants of the Submesh.
are not contiguous in the array, forcing a movement of these data blocks to a user buffer. Unless these data movements are overlapped with communication, this is an additional penalty paid by CE for exchanging two quadrants at a time. (Algorithms like Quadrant Exchange [10], which only exchange one quadrant at a time do not have this overhead.) CE achieves a complete overlap of data block movement with communication by a combination of proper arrangement of quadrants in different processors and the use of four types of exchange patterns (see Figure A.1). This mechanism, described further below, leads to Level 2 Cyclic Exchange.

The sequence of data quadrants within the array is different for white and black nodes. For white nodes, data quadrants are organized in the following order: upper left (quadrant numbered 0), lower left (quadrant 1), upper right (quadrant 2), and lower right (quadrant 3). For black nodes, the order is: upper left (quadrant 0), upper right (quadrant 1), lower left (quadrant 2), and lower right (quadrant 3). Since whiteness and blackness of nodes remains invariant in all phases except the last \( i = 0 \), two kinds of reorder schemes are needed: White-to-White and Black-to-Black. In the last phase, all nodes are Whites; so black nodes will perform a Black-to-White reorder at the end of phase \( i = 1 \). It is assumed that every processor has its data in the appropriate format when the complete exchange begins. If they don't, the time of reordering before the start of the algorithm can be reduced by a scheme which is described further below.

The algorithm also uses another data buffer in user space, of size \( \frac{3}{4} \cdot 2^{2p} m \) (3 quadrants). Consider, for example, the communications of a white node in the upper left quadrant of the submesh (which varies from phase to phase). It needs to send the data blocks in data quadrants UR and LR (numbered 2 and 3) in its first (horizontal)
algorithm CyclicExchange(r, c).
begin
for i ← p - 1 downto 0 do { Phase i }
  HorizProc ← (r, c ± 2^i)
  VertProc ← (r ± 2^i, c)
  if ( r mod 2 = c mod 2 ) or ( i = 0 ) {White Node}
    Partner1 ← HorizProc; Partner2 ← VertProc
  else {Black Node}
    Partner2 ← HorizProc; Partner1 ← VertProc
  endif
  Step1: Send 2^{2p-1} data blocks (2 quadrants) to Partner1
    Post a receive from Partner1. with user buffer as destination
    Gather a data quadrant into user buffer
    Gather a quadrant into the data array.
      from the old copy of the data array
    Complete the receive from Partner1
  Step2: Exchange 2^{2p-1} data blocks with Partner2
    Send 2^{2p-1} data blocks (2 quadrants) to Partner2
    Post a receive from Partner2. with user buffer as destination
    Gather a data quadrant into user buffer
    Gather a quadrant into the data array.
      from the old copy of the data array
    Complete the receive from Partner2
    if ( i > 0 ) perform interphase data reordering of 2^{2p-1}m bytes
  endfor
end

Figure A.2: The Cyclic Exchange Algorithm for Complete Exchange.
communication (see Figure A.1. It receives the data quadrants UL' and LL' from its horizontal counterpart. If these were placed in the data array directly, the data quadrants containing LL and LL' are not contiguous, necessitating data movement for the next (vertical) communication. So, CE places the incoming data quadrants in a user buffer. While the first (horizontal) communication is progressing, the data quadrant LL is moved to the user buffer (see Figure A.1) so that it is contiguous with LL'. The vertical send is now initiated from the user buffer, with the received data being placed in the two data quadrants that were used in the previous (horizontal) communication. Concurrent with the vertical communication, the data quadrant UL' in the user buffer is moved to the space occupied by LL earlier in the data array.

To reduce the cost of reordering, our implementation of Cyclic Exchange uses two copies of the data array, and toggles between them in successive phases. This allows each processor to reorder only that half of the array which is needed for the first communication in the next phase. The other half is reordered as the first communication of the next phase progresses. This cuts the non-overlapped component of reordering in half. This addition, which we shall call Level 3, implies that the dashed arrows in Figure A.1 really refer to placement of the necessary data blocks in the user buffer from the old copy of the data array. Level 3 is the complete version and its algorithm is listed in Figure A.2. Note that this scheme can also be used to reduce the initial reordering cost if the data is not in the appropriate format when the algorithm begins.

Thus, the space complexity of Cyclic Exchange is \(11 \cdot 2^{p-2}m\). To see that this is a modest requirement, consider that for a 32 \(\times\) 32 mesh \((p = 5)\) with \(m = 1024\) bytes, the storage required is less than 3 MB, whereas most modern systems provide 16 MB or more per node.
APPENDIX B

IMPLEMENTATION ISSUES IN CRAY T3D

Certain issues had to be handled in implementing the hybridization scheme on the T3D. The performance had to be measured for at least an $8 \times 8$ mesh since subtle issues involving contention do not arise in smaller meshes. Yet, since this installation had a topology of $8 \times 4 \times 4$ nodes, it is not possible to get a physically square $8 \times 8$ partition. This necessitates an embedding of the logically square mesh onto the torus structure. Such an embedding should ideally preserve the contention pattern that would be seen on a physically square mesh, but this is very difficult to do with Direct Exchange (DE) since its contention pattern is hard to characterize, especially in the absence of barrier synchronization between steps. Since Cyclic Exchange (CE) has a predictable pattern of contention, an embedding was chosen so as to minimize the deviation from the pattern of contention that would be seen on a square mesh for Cyclic Exchange. This embedding is done as follows. Number the nodes in the logical square mesh in row-major order and those in the physical machine also in row-major ($xyz$) order. The mapping from logical node identifiers ($i_l$) to the physical node numbers ($i_p$) is:

$$i_l \rightarrow i_p = \begin{cases} 
  i_l & \text{if } i_l \text{ in } 0\ldots31 \\
  i_l + 32 & \text{else}
\end{cases}$$

131
This embedding has the property that horizontal communications in CE will have
the same contention pattern as with a physically square mesh. The vertical com-
munications of CE in the initial phase will, however, face no contention unlike the
pattern in the square mesh. The pattern of contention in subsequent phases will be
exactly identical to that in a square mesh. Also, CE will never use the wrap-around
meshes of the torus, although the same cannot necessarily be expected of DE. It is
to be noted that the embedding is necessary only for the $8 \times 8$ mesh: it is possible to
acquire partitions that are physically $2 \times 2$ or $4 \times 4$. 


133


