Compile-time and Run-time Strategies for Array Statement Execution on Distributed-Memory Machines

Dissertation

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

By

Shivnandan D. Kaushik, B.Tech., M.S.

***

The Ohio State University

1995

Dissertation Committee:
Prof. C.-H. Huang
Prof. P. Sadayappan
Prof. D. K. Panda
Prof. M. Singhal

Approved by
Co-Adviser
Co-Adviser
Department of Computer and Information Science
To my family
ACKNOWLEDGEMENTS

I would like to thank my advisors Prof. Chua-Huang Huang and Prof. P. Sadayappan for providing me with the proper stimulus at the start of my graduate program, for giving me timely constructive criticism and feedback throughout the course of the program, for being ever willing to listen to ideas and for teaching me to build on them, and for spending enormous amounts of their time on our work in general and on this dissertation in particular. It has been a privilege to have known them and a pleasure to have worked with them.

I am grateful to Prof. Dhabaleshwar Panda and Prof. Mukesh Singhal for serving on my dissertation committee. I would like to thank Prof. Dhabaleshwar Panda for his seminars on interconnection networks which have had a major influence on this work. Prof. Mukesh Singhal has been a source of advice from time to time. For this I am grateful to him. I also thank Prof. Mukesh Singhal and Prof. Steve Lai for playing the important roles of my minor advisers. Prof. Steve Lai has been a source of encouragement always exhorting me to hasten my graduation. I thank Prof. Robert Johnson at St. Cloud State University, Prof. Jeremy Johnson at Drexel University, and Prof. J. Ramanujam at Louisiana State University for many discussions on our collaborative work. I would also like to thank Stuart Goossen at Intel Corporation for his interest in my work.
I was supported by ARPA and a University Presidential Fellowship during the course of my graduate study at The Ohio State University. I thank ARPA and Graduate School at The Ohio State University for the same.

My numerous colleagues at Ohio State have made my stay in this town an enjoyable and learning experience. Sandeep Gupta has been a friend and partner in research. I learned a lot through our discussions. I have enjoyed working with Bharat Kumar, Jinghuan Lu, Saad Mufti, Sanjay Sharma, and Raj-Vardhan Singh. Jayant Ananthraman, Davendar Babbar, Debhasis Basak, Paolo Bucci, Jim Clausing, Bharat Kumar, Manas Mandal, Ravi Prakash, Ashutosh Sabharwal, and Sunil and Jyothi Thandani have been great company. Special thanks to Manas Mandal for providing and updating the dissertation style file and maintaining a store of health food in his office. Special thanks to Debhasis Basak for many discussions on issues related to parallel architectures. Several friends have provided me with moral support on many occasions. I cannot find the words to thank Anuradha Bhave, Sanjay and Sona Sharma, and Varad Joshi. Manish Prabhu has been a close friend and a great roommate. I thank him for putting up with my irregularities and a lot more.

My siblings, Shantesh, Aparna, Swati, and Kaushal have always inspired me through their achievements in academic and non-academic activities. I would like to thank them for their guidance and affection. It is difficult to express in words my gratitude towards my parents for their love, support, and the sacrifices they have made for me. My family has always been there for me through good and bad times. It is to them that I dedicate this thesis.
VITA

March 15, 1969 ................................................... Born - Basrah, Iraq

1990 ................................................... B.Tech. Computer Science and Engg,
Indian Institute of Technology,
Bombay, India

1992 ................................................... M.S. Computer Science,
The Ohio State University

1990-1991 .............................................................. Graduate Teaching Associate,
The Ohio State University.

1991-1994 .............................................................. Graduate Research Associate,
The Ohio State University.

1994-1995 .............................................................. Presidential Fellow,
The Ohio State University.

Publications

S. D. Kaushik, S. Sharma, C.-H. Huang. An algebraic theory for modeling multistage
interconnection networks. Journal of Information Science and Engineering, 1–26,

S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase re-
distribution: Modeling and evaluation. In Proc. of International Parallel Processing

Johnson, and P. Sadayappan. A Portable Programming Environment for Designing
and Implementing High-Performance Block Recursive Algorithms In Proc. Supercom-


Fields of Study

Major Field: Computer and Information Science

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Professors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Architecture</td>
<td>C.-H. Huang, P. Sadayappan</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>M. Singhal</td>
</tr>
<tr>
<td>Algorithms</td>
<td>S. Lai</td>
</tr>
</tbody>
</table>
# Table of Contents

**DEDICATION** ................................................................. ii

**ACKNOWLEDGEMENTS** ................................................................. iii

**VITA** .................................................................................... v

**LIST OF TABLES** ................................................................. xiii

**LIST OF FIGURES** ................................................................. xvi

**CHAPTER** ............................................................. PAGE

I  Introduction ........................................................................... 1
  1.1 Background and Motivation .................................................. 1
  1.2 Problem Statement and Our Approach .............................. 6
    1.2.1 Array Statement Execution ........................................ 6
    1.2.2 Array Redistribution ............................................... 11
  1.3 Related Work ................................................................. 14
    1.3.1 Array Statement Execution ........................................ 14
    1.3.2 Array Redistribution ............................................... 21
  1.4 Organization of Thesis .................................................... 24

II  Array Statement Execution on Distributed-Memory Machines .......................... 25
  2.1 Alignment and Distribution Directives .............................. 26
  2.2 Array Assignment Statements .......................................... 33
  2.3 Array Statement Execution Using Array Scanning ............. 38
  2.4 Array Statement Execution Using Index Sets ..................... 41
  2.5 Summary ........................................................................ 46
## III Index Sets for One-Level Mappings

3.1 Local Index Sets for the Array Statement
   3.1.1 Block Distribution
   3.1.2 Cyclic Distribution
   3.1.3 Modifications for Negative Strides

3.2 Communication Sets for the Array Statement
   3.2.1 Block Distribution to Block Distribution
   3.2.2 Block Distribution to Cyclic Distribution
   3.2.3 Cyclic Distribution to Block Distribution
   3.2.4 Cyclic Distribution to Cyclic Distribution
   3.2.5 Modifications for Negative Strides

3.3 Virtual Processor Approach for Block-Cyclic Distributions
   3.3.1 Virtualization Views
   3.3.2 Virtualization Schemes
   3.3.3 Strategy for Selection of Virtualization Schemes

3.4 Index Sets for Multi-Dimensional Arrays

3.5 Performance Results

3.6 Discussions

3.7 Summary

## IV Index Sets for Two-level Mappings

4.1 Mathematical Preliminaries

4.2 Block and Cyclically Distributed Templates
   4.2.1 Memory Allocation
   4.2.2 Index Set Translation
   4.2.3 Block Distributed Template
   4.2.4 Cyclically Distributed Template

4.3 Virtual Processor Approach for Two-Level Mappings
   4.3.1 Virtual Block View
   4.3.2 Virtual Cyclic View
   4.3.3 Strategy for Selection of Virtualization Schemes

4.4 Performance Results

4.5 Summary
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Index Sets for Array Redistribution</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>5.1 Array Redistribution</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>5.2 The Tensor Product Representation</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>5.3 A Mixed Radix Representation of Regular Data Distributions</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>5.3.1 Representation of One-Dimensional Arrays</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>5.3.2 Representation of Multi-Dimensional Arrays</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>5.3.3 Array Redistribution</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>5.3.4 Index Set Representation</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>5.4 Index Sets for Array Redistribution</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>5.4.1 Array Redistribution of One-Dimensional Arrays</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>5.4.2 Array Redistribution of Multi-dimensional Arrays</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>5.5 Summary</td>
<td>157</td>
</tr>
<tr>
<td>VI</td>
<td>Multi-Phase Array Redistribution</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>6.1 A Communication Cost Model for Array Redistribution</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>6.1.1 Array Redistribution of One-Dimensional Arrays</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>6.1.2 Array Redistribution of Multi-dimensional Arrays</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>6.2 An Indexing Cost Model for Array Redistribution</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>6.3 A Multi-Phase Approach for Array Redistribution</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>6.3.1 Array Redistribution from a Cyclic(Yt) to a Cyclic(t) Distribution</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>6.3.2 Array Redistribution from a Cyclic(s) to a Cyclic(t) Distribution</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>6.4 Performance Results</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>6.5 Summary</td>
<td>186</td>
</tr>
<tr>
<td>VII</td>
<td>Conclusions</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>7.1 Summary of Principal Contributions</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>7.2 Directions for Future Research</td>
<td>191</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>Proof of Optimality of Min-Tuple for Y = r^k</td>
<td>193</td>
</tr>
</tbody>
</table>
B Heuristics for General Y ........................................ 198

BIBLIOGRAPHY ....................................................... 200
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Index mapping for regular data distributions. $N = n - m + 1$</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>Virtualization schemes.</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>Send and receive processor set size estimates for various source and</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>target distributions.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Evaluation of the strategy for the virtualization scheme selection $(P =</td>
<td>83</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation of the strategy for the virtualization scheme selection $(P =</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>Evaluation of the strategy for the virtualization scheme selection $(P =</td>
<td>85</td>
</tr>
<tr>
<td>7</td>
<td>Comparison of the Execution Time and the Table Generation Time $(P = 32,</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>$M = 230400)$.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Comparison of the Execution Time and the Table Generation Time $(P = 32,</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>$M = 230400)$.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Comparison of the Execution Time and the Table Generation Time $(P = 32,</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>$M = 230400)$.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Comparison of table generation times for block distributed template under</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>a virtual block view. $P = 32, M = 23040$.</td>
<td></td>
</tr>
</tbody>
</table>
11 Comparison of table generation times for indexing schemes for cyclic(720) distributed template using virtual block view. $P = 32, M = 23040$.

12 Comparison of table generation times for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$.

13 Comparison of table generation times for indexing schemes for cyclic(720) distributed template using virtual cyclic view. $P = 32, M = 23040$.

14 Comparison of table generation times for indexing schemes for cyclic(80) distributed template using virtual cyclic view. $P = 32, M = 23040$.

15 Comparison of execution times for block distributed template under virtual block view. $P = 32, M = 23040$.

16 Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$.

17 Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual cyclic view. $P = 32, M = 23040$.

18 Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual cyclic view. $P = 32, M = 23040$.

19 Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual cyclic view. $P = 32, M = 23040$.

20 Comparison of memory wasted for block distributed template under the virtual block view. $P = 32, M = 23040$.

21 Comparison of memory wasted for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$. 

22 Comparison of memory wasted for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$. 

23 Comparison of memory wasted for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$. 

xiv
24 Memory allocation time for allocation schemes with hole-compression for block distributed template using virtual block view. $P = 32, M = 23040$. ................................................................. 133

25 Memory allocation time for allocation schemes with hole-compression for cyclic(720) distributed template using virtual block view. $P = 32, M = 23040$. ................................................................. 133

26 Memory allocation time for allocation schemes with hole-compression for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$. ................................................................. 133

27 Memory allocation time for allocation scheme with hole-compression for cyclic(720) distributed template using virtual cyclic view. $P = 32, M = 23040$. ................................................................. 134

28 Memory allocation time for allocation scheme with hole-compression for cyclic(80) distributed template using virtual cyclic view. $P = 32, M = 23040$. ................................................................. 134

29 Memory allocation time for allocation scheme with hole-compression for cyclic distributed template using virtual cyclic view. $P = 32, M = 23040$. ................................................................. 134

30 Index mapping functions for regular data distributions. ......... 142

31 Indexing cost measures for special forms of array redistribution. ... 170

32 Indexing cost measures for the two-phase array redistribution. ... 175

33 Execution times (ms) for cyclic(s) to cyclic(t) redistribution on 32- processor Cray T3D. ................................................................. 185

34 Execution times (ms) for cyclic(s) to cyclic(t) redistribution on 64- processor IBM SP2. ................................................................. 187
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>45</td>
</tr>
</tbody>
</table>

*The HPF two-level mapping model for data arrays.*

*HPF code segment illustrating array alignment.*

*HPF array alignment illustration.*

*HPF array distribution illustration.*

*Memory allocation for array mapped using a two-level mapping with stride (a) Layout of template cells among processors. (b) Local array without hole-compaction. (c) Local array with hole compaction.*

*Illustration of execution of array statement $B(0:12:2) = f(A(0:18:3))$ for arrays $A(0:23)$ and $B(0:19)$.*

*Illustration of execution of array statement $B(0:12:2) = f(A(0:18:3))$ for block distributed arrays $A(0:23)$ and $B(0:19)$ on four processors.*

*Array assignment execution using array scanning.*

*Array statement execution using index sets.*

*Variable stride in $A.loc$ for global array section $A(0:40:2)$.*

*Memory allocation for an array mapped using a two-level mapping with stride.*
Array statement execution involving multiple right hand side arguments. 46
Closed form for local index set for block distributed array. 50
Closed forms for block to block case. 54
Node code for $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ - Sending phase. $A(m_1 : n_1)$ and $B(m_2 : n_2)$ are block distributed on $P_1$ and $P_2$ processors, respectively. 57
Node code for $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ - Receive Phase. $A(m_1 : n_1)$ and $B(m_2 : n_2)$ are block distributed on $P_1$ and $P_2$ processors, respectively. 58
Closed forms for block to cyclic case ($P_2 = 3, s_2 = 2$). 60
Stride distances for $PRecv(p)$ for block to cyclic case ($P_2 = 3, s_2 = 2$). 61
Determination of $DRecv(p, q)$ for block to cyclic case. 62
Virtual views of a block-cyclic distribution. 67
Virtual-block view of array $A(0 : 15)$ with cyclic(2) distribution on two processors. 70
Virtual-cyclic view of array $A(0 : 15)$ with cyclic(2) distribution on two processors. 71
Communication phases in the virtual processor domain. 75
Template for node code with virtual processor approach: Send Phase. 76
Template for node code with virtual processor approach: Receive phase. 77
Send phase for two-dimensional arrays in the virtual processor domain. 82
Receive phase for two-dimensional arrays in the virtual processor domain. 82
Execution of program with explicit computation distribution. 89
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Execution of program with explicit computation distribution</td>
<td>90</td>
</tr>
<tr>
<td>30</td>
<td>Memory allocation for array mapped using a two-level mapping with non-unit stride viewed under the virtual-cyclic view</td>
<td>95</td>
</tr>
<tr>
<td>31</td>
<td>Functions for translating between template sections and array sections.</td>
<td>102</td>
</tr>
<tr>
<td>32</td>
<td>Methodology for determining size of local array with hole compression.</td>
<td>103</td>
</tr>
<tr>
<td>33</td>
<td>Methodology for index set translation.</td>
<td>104</td>
</tr>
<tr>
<td>34</td>
<td>Index Set Translation on processor $p$.</td>
<td>106</td>
</tr>
<tr>
<td>35</td>
<td>Memory allocation for array mapped using a two-level mapping with non-unit stride.</td>
<td>111</td>
</tr>
<tr>
<td>36</td>
<td>Two-dimensional Local Memory Allocation for array mapped using a two-level mapping with non-unit stride under virtual block view.</td>
<td>114</td>
</tr>
<tr>
<td>37</td>
<td>Two-dimensional Local Memory Allocation for array mapped using a two-level mapping with non-unit stride under the virtual cyclic view.</td>
<td>117</td>
</tr>
<tr>
<td>38</td>
<td>Memory allocation for array mapped using a two-level mapping with non-unit stride.</td>
<td>118</td>
</tr>
<tr>
<td>39</td>
<td>HPF program segment.</td>
<td>142</td>
</tr>
<tr>
<td>40</td>
<td>Redistribution of array $A(0:14)$ from block to cyclic distribution.</td>
<td>143</td>
</tr>
<tr>
<td>41</td>
<td>Array redistribution using index sets.</td>
<td>144</td>
</tr>
<tr>
<td>42</td>
<td>Loop nest for enumerating index set $e^C_x \otimes e^Y_y \otimes e^b_z$, $y \in (0: Y - 1 : P)$.</td>
<td>150</td>
</tr>
<tr>
<td>43</td>
<td>$Cyclic(Yt)$ to cyclic($t$) redistribution: Construction of $PSend(p)$ for $Y \geq P$.</td>
<td>152</td>
</tr>
<tr>
<td>44</td>
<td>Node program on processor $(p_{t-1}, \ldots, p_0)$ for multi-dimensional array redistribution.</td>
<td>158</td>
</tr>
</tbody>
</table>
First destination processors for array redistribution of \( A(96) \) from \( \text{cyclic}(12) \) to \( \text{cyclic}(2) \) on eight processors. ........................................ 163

Distributed algorithm for performing communication for \( \text{cyclic}(Yt) \) to \( \text{cyclic}(t) \) redistribution. ....................................................... 166

Node code for packing data for array redistribution: node program for processor \( p \). ................................................................. 169

Execution times for \( \text{cyclic}(240) \) to \( \text{cyclic}(8) \) and \( \text{cyclic}(192) \) to \( \text{cyclic}(8) \) redistributions on 32 node T3D. ........................................ 177

Execution times for \( \text{cyclic}(3000) \) to \( \text{cyclic}(50) \) and \( \text{cyclic}(1000) \) to \( \text{cyclic}(50) \) redistribution on 64 node IBM SP2. ................................. 180

Execution times for \( \text{cyclic}(8) \) to \( \text{cyclic}(240) \) and \( \text{cyclic}(8) \) to \( \text{cyclic}(192) \) redistribution on a 32 node T3D. ........................................ 182

Execution times for \( \text{cyclic}(50) \) to \( \text{cyclic}(3000) \) and \( \text{cyclic}(50) \) to \( \text{cyclic}(1000) \) redistribution on 64 node IBM SP2. ................................. 184

Min-Tuple \( m(44, 8) = (6, 6, 6, 6, 5, 5, 5, 5) \) ........................................ 195

Relation between tuple \( t \) and min-tuple \( m \). ........................................ 196
CHAPTER I

Introduction

1.1 Background and Motivation

As processor clock speeds and packaging technology reach their inherent physical limits, parallel computing represents the only plausible direction to continue increasing the computational power available to computational scientists and engineers. However, the use of parallel computers is not likely to be a success unless these computers are as easy to use as the previous generation of vector supercomputers such as the Cray X-MP [61, 71]. The success of vector supercomputers can be largely attributed to the identification of a sub-dialect of Fortran to write machine-independent vector programs. Advances in compiler technology, specifically techniques for automatic vectorization [3, 98, 134, 135, 137, 139], had made possible the development of a set of rules for vectorizing loops in Fortran programs. After application of these well understood rules to a Fortran program, translation of that program to an efficient vector program for any vector machine could be achieved.

However, the situation with current parallel machines is different. While programming such a machine, a programmer has to modify an existing program and extend it using language constructs which explicitly reflect the architecture of the underlying
parallel machine. To obtain high sustained performance, it is necessary to tune the program, taking into account the idiosyncrasies of the underlying architecture, such as the communication protocols and memory hierarchy. This program tuning is specific to the particular machine and the time and effort invested is often wasted as the tuned program is not guaranteed to achieve high performance on a different parallel machine or even the next release of the same parallel machine.

Current parallel machines can be classified into shared-memory multiprocessors and distributed-memory multiprocessors. Shared-memory multiprocessors such as the BBN Butterfly, the Silicon Graphics Challenge, and the Cray Y-MP have a global memory which is accessible by every processor [60, 61]. Programming such machines is relatively simple as the program can still be written in a single address space as in a sequential program. Additional constructs for identifying statements which can be executed in parallel and for performing synchronization among the processors of the parallel machine are provided in the language extensions for these machines. However, the provision of the global memory comes at the expense of complex hardware. Hence these machines cannot be scaled to a large number of processors.

On the other hand, most modern distributed-memory multiprocessors such as the IBM SP1/SP2 [51, 119], Intel Paragon XP/S [69], Cray T3D [38], Intel iPSC/860 [70], and Thinking Machines CM-5 [22, 37] are constructed by connecting off-the-shelf processors using an interconnection network. The scalability of the system is only restricted by the scalability of the network. Commercial distributed-memory machines with thousands of processors have been constructed. In such a distributed-memory
architecture, each processor has its own local memory and does not have direct access to another processor's memory. The programmer exploits parallelism by distributing the data among the local memories of the processors and partitioning the computation among the processors. The programming style follows an SPMD (single program multiple data) model. In this model, the same program executes on all the processors but operates on different data in the processors' memory. Whenever a processor requires data that is not located in its local memory, the data is obtained by performing explicit interprocessor communication. The processor on which the required data is located performs an explicit message send to the requesting processor; while the requesting processor performs an explicit message receive. Thus in writing an SPMD program, the programmer has to determine points in the program at which communication will be required and insert the appropriate communication commands. This is a cumbersome and error-prone task. Thus although distributed-memory machines have potential for providing high-performance, achieving a high sustained software performance is often difficult. The programming effort involves identifying non-obvious tradeoffs between several possible data and computation distributions. The programmer has to work with a disjoint address space using the message passing paradigm. This process is usually time-consuming, error-prone and restricts the programmer from experimenting with different data and computation distributions.

A problem widely addressed in the language and compiler community for parallel computing, has been the development of compiler techniques for starting with a sequential program, identifying the best data and computation partition for a
distributed-memory machine, and automatically generating the corresponding SPMD program. A consensus reached has been that the development of such compiler techniques is in general a difficult problem. While several methods for performing automatic data decomposition and data alignment [8, 9, 10, 12, 11, 32, 33, 34, 35, 36, 50, 53, 54, 56, 50, 67, 79, 80, 89, 87, 88, 102, 103, 104, 132, 133] have been presented in the literature, they are applicable only to specific forms of Fortran programs. An intermediate goal set up by the community was that of identifying additional information which a programmer with a clear understanding of the application could provide to the compiler to facilitate efficient compilation for distributed-memory machines. This separation of concerns would allow a two-pronged effort for developing an efficient portable programming environment for these machines. The two independent efforts would be as follows.

- Development of compilation techniques for distributed-memory machines for programs annotated with additional information provided by the programmer. The compiler would start with a program in the Fortran sub-dialect with the additional programmer-supplied annotations and generate a message passing SPMD program.

- Development of compiler techniques for identifying the requisite additional information for un-annotated programs. These techniques would start with an un-annotated program in the sub-dialect and generate the necessary annotations. This annotated program can then be translated into an SPMD program by the methods developed for the previous problem.
It has been recognized that identifying the data decomposition is one of the most difficult and intellectually challenging step in the development of large parallel programs. Furthermore, the provision of a single address space has also been recognized as an important feature for the programmer. Several languages such as Fortran D [47], CM-Fortran [116], pC++ [13], PANDORE [43], MPP Fortran [100], and Vienna Fortran [29, 28, 27] have been defined to be portable parallel programming environments for distributed-memory machines. These languages provide the programmer with a single address space and allow annotations to specify the mapping of data arrays to the processors of the machine. The regular data distributions supported by these languages are the block, the cyclic, and the block-cyclic distribution. The definition of a common base language - High Performance Fortran (HPF) [46, 86, 84, 90] - which selects features of the above-mentioned languages, is available. Several commercial compilers for “Subset HPF” - a language subset defined to facilitate early completion of the first HPF compilers - are available. When programming in HPF, the programmer writes a single address space program with a single thread of control and provides information for mapping arrays to the processors. When compiling a program written in HPF for a distributed-memory machine, it is the responsibility of the compiler to partition the arrays among the processors and generate the required communication commands to transfer data between the local memories of the processors at appropriate points in the execution of the parallel program. The development of compilation techniques for translating HPF programs to efficient message passing node programs is important. Some features in HPF were deemed to be difficult for
efficient implementation and were not included in "Subset HPF". One of the issues which was not addressed in "Subset HPF" was the efficient execution of array statements and dynamic array redistribution on distributed-memory machines. In this thesis, we develop compiler techniques for the execution of array statements and for performing array redistribution in HPF-like languages.

1.2 Problem Statement and Our Approach

1.2.1 Array Statement Execution

The array assignment statement in Fortran 90 [1, 68, 93, 95] is provided for expressing simultaneous assignments to large groups of array elements. The ability to express multiple assignments in a single statement provides a convenient syntax for expressing data parallelism. Array assignment statements involve array sections which consist of array elements from a lower index to an upper index at a fixed stride. In order to generate high-performance target code, HPF compilers for distributed-memory machines should produce efficient code for array statements involving distributed arrays.

Compilation of array statements with distributed arrays for parallel execution requires partitioning of the computation among the processors. Many compilers use the owner-computes rule [111, 101, 112] to partition the computation in such a manner, that the processor to which the array element of the distributed array is mapped performs the computation which modifies it. The computation performed on a processor may involve array elements resident on other processors. In the generated
code, all the non-local data needed by a processor is fetched into temporary arrays in a processor’s local memory using interprocessor communication. Distributed-memory machines are characterized by high communication overheads for interprocessor communication. In order to reduce the communication overhead, each processor first determines all the data it needs to send to and receive from other processors and then performs the needed communication. This reduces the communication overhead by aggregating all the data movements needed from one processor to another into a single message. After communication, each processor then performs the computation in its local address space. Thus, the execution of the array statement incurs both a communication and an indexing overhead.

The indexing overhead to perform the data movement for the execution of the array statement consists of the determination of the following local index sets, data index sets, and processor sets for each processor \( p \):

- **Local index set** of processor \( p \): indices of the array elements which are resident on \( p \) and for which \( p \) must compute new values as would be evaluated by the array assignment statement.

- **Send processor set** of processor \( p \): set of processors to which \( p \) has to send data.

- **Send data index set** of processor \( p \) to processor \( q \): indices of the array elements which are resident on \( p \) but are needed by \( q \).

- **Receive processor set** of processor \( p \): set of processors from which \( p \) has to receive data.
• Receive data index set of processor \( p \) from processor \( q \): indices of the array elements which are needed by \( p \) but are resident on \( q \).

A parameterized closed form characterization of these sets would reduce the overhead of packing data into messages on the sending processor and unpacking data at the receiving processor. If the arrays have only block or cyclic distributions, then the data index sets and the processor sets can be characterized using regular sections for closed forms [58, 59, 82]. For many algorithms, for instance, the solution of a dense linear system of equations, the best data distributions are block-cyclic distributions since they provide a good tradeoff between reducing communication requirements and achieving good computational load balance [41]. However, for the general block-cyclic distribution, closed form characterization of the index sets using simple regular sections is not possible. The implementation of the indexing routines for block-cyclic routines has been considered problematic. For instance, since it was believed that the efficient compilation of array statements with non-unit strides involving block-cyclically distributed arrays was not feasible, Cray's T3D compiling system [100] imposes the severe restriction that the block size of the distributed array and the number of processors along the distributed dimension be a perfect power of two [91] so that bit manipulation can be used for performing address computation and determining interprocessor communication.

In this thesis, we present a virtual processor approach to efficiently enumerate the data index sets and processor sets when arrays have block-cyclic distributions. This approach is based on viewing a block-cyclic distribution as a block (or cyclic)
distribution on a set of virtual processors, which are cyclically (or block-wise) mapped to physical processors. These views are referred to as virtual-block or virtual-cyclic views depending on whether a block or cyclic distribution of the array on the virtual processors is used. These virtualization views permit us to use the closed forms for block and cyclic distributions in the virtual processor domain. A processor performs the computation for the virtual processors mapped to it. A message from processor \( p \) to \( q \) consists of all the data to be sent from the virtual processors on \( p \) to the virtual processors on \( q \).

Under the owner-computes rule, a processor owning the array element in an array section on the right hand side of the array assignment statement sends data to the processor owning the corresponding element of the array section on the left hand side of the assignment. We call the array section which is assigned the value as the target array section and an array section of the array expression on the right hand side of the array assignment statement as a source array section. For an array statement, each source and target array section pair is analyzed to determine the total data to be sent and received by each processor. The indexing overhead for a source and target array section pair depends upon the virtualization views used for each source and target array dimension. There are four possible combinations of virtual views per dimension as either a virtual block or a virtual cyclic view can be used for each array axis. Although each of the possible combinations involves exactly the same communication between any pair of physical processors, they will generally have different indexing overhead. We present a selection strategy to choose the virtualization scheme with
the minimum indexing overhead.

We have implemented the virtual processor approach on a Cray T3D multicomputer. The performance measurements demonstrate that the indexing overheads for the four virtualization schemes are significantly different. The selection strategy provides a good prediction of the virtualization scheme with the lowest indexing cost.

The index set enumeration for block-cyclic distributions is further complicated due to the two-level mapping scheme supported by HPF for mapping data arrays to an abstract processor grid. The language introduces a Cartesian grid referred to as a template. Arrays are aligned to the template and the template is distributed onto a processor array using regular data distributions. The alignment of the array elements to the template cells can be performed at a stride with an initial offset. For instance, an array $D(0 : N - 1)$ can be aligned with a template $T$ such that the array element $D(i)$ is aligned with template cell $T(ci + a)$, where $c$ is the alignment stride and $a$ is the alignment offset. If memory for the portion of the distributed array mapped to a processor is allocated for all the template cells mapped to the processor, then the introduction of the stride $c$ in the array-template mapping creates "holes" in the allocated array on the processor as some template cells do not have any array elements mapped to them. These holes represent memory which will never be utilized and hence should be removed by allocating memory only for those template cells with which array elements are aligned. This "hole-compression" complicates the enumeration of the local index sets and the send and receive data index sets.

Our approach for addressing the memory allocation and indexing problems for
arrays mapped using two-level mappings is as follows. For block and cyclically distributed templates, using the regular section characterization of the template cells mapped to a processor \( p \), we develop a parameterized regular section characterization of the array elements located on \( p \). This characterization facilitates the determination of the exact amount of memory to be allocated for the distributed array on processor \( p \). For block and cyclically distributed arrays, the data index sets and local index sets for the array statement in template index space, can be expressed as simple regular sections of the uncompressed local template array. We show that for block and cyclically distributed templates, a regular section in the uncompressed local template array corresponds to a regular section in the compressed local array and develop a strategy for performing the translation between the local template section and the local array section. Using this procedure, the parameterized closed forms for the send and receive data index sets and the local index sets for one-level mappings can be extended to those for the two-level mappings. Using the results for the block and cyclically distributed arrays, the virtual processor approach for handling block-cyclic distributed templates is extended. The key idea in the extension is to look upon the local array as a two-dimensional array with one dimension corresponding to the virtual processor index and the other corresponding to the local index space of the virtual processor.

1.2.2 Array Redistribution

Array redistribution is a special form of the array statement and involves changing the data distribution of a distributed array. Efficient distributed-memory implemen-
tations of several scientific applications including the multi-dimensional fast Fourier transform [127] and the Alternate Direction Implicit (ADI) [130] method for solving two-dimensional diffusion equations require array redistribution. Array redistribution is also required when the distributed arrays passed as arguments to runtime libraries or subroutines have distributions different from those of the formal parameters. Thus efficient methods for performing array redistribution are of great importance for the development of distributed-memory compilers for HPF-like languages.

Similar to the array statement, performing array redistribution incurs an indexing and communication overhead. The indexing overhead arises due to the enumeration of the processor and data index sets. The communication overhead arises from the all-to-many personalized communication among the processors. A processor communicates with several other processors, sending distinct array elements to each processor. The cost for performing the communication includes the message startup and transmission costs and the overhead arising due to node and link contention. For worm-hole routed and circuit-switched networks, the cost of transmitting a large message is nearly independent of the distance between the source and target nodes [39, 94]. Node contention arises as a processor can typically receive a single message at a time. Hence, if a target node has to receive messages from more than one source node at a time, then one of the source nodes has to wait to perform the communication. Link contention arises when two or more message paths need to share the same interconnection network edge. The transmission of the messages in this case will be sequentialized (especially for circuit-switched networks).
The virtual processor approach for the execution of array statements can also be used for performing the indexing for array redistribution. This approach would require additional memory and computation at run-time for efficiently enumerating the index sets. However, array redistribution involves all the elements of the distributed array and the number of variable parameters for array redistribution are smaller than those for the execution of the array statement, i.e., array redistribution is a special case of the array assignment statement. Furthermore, the all-to-many personalized communication due to array redistribution might have some identifiable patterns. So the questions concerning the development of efficient redistribution routines are as follows.

1. Is it possible to develop methods for performing array redistribution which are more efficient than those for execution of the general array statement?

2. Is it possible to use case-specific information to schedule and reduce the communication cost for the all-to-many personalized communication for array redistribution?

In this thesis, we develop precise closed forms for enumerating the processors to communicate with, and the array elements to be communicated for two special cases of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. Using the developed closed forms, a distributed algorithm for scheduling the communication for array redistribution is developed. The generated schedule eliminates node contention and incurs the least communication
overhead. The scheduling algorithm has an asymptotically lower scheduling overhead than techniques presented in the literature. Based on the developed closed forms, a cost model for estimating the communication and the indexing overhead for array redistribution is developed. Using this model, a multi-phase approach for reducing the communication cost for array redistribution is presented. The key idea is to perform the redistribution as a sequence of redistributions such that the total cost of the sequence is less than that of direct redistribution. Algorithms for determining the sequence of intermediate data distributions which minimizes the total redistribution time are developed. Experimental results on the Cray T3D and IBM SP2 demonstrate the validity of the developed cost models and the efficacy of the multi-phase approach for array redistribution.

We now describe work presented in the literature for the efficient execution of the array statement and array redistribution on distributed-memory machines.

1.3 Related Work

1.3.1 Array Statement Execution

The issue of generating code for message passing machines from a single address space program with data distribution directives was addressed by Koelbel in the context the programming language KALI [82, 83, 85]. A closed form characterization of the data index sets was provided for computations involving identically distributed arrays with either a block or a cyclic distribution. Regular sections with unit stride were considered in [85], while non-unit-stride regular sections were considered in [82,
Closed form characterizations of the processor sets were not developed and arrays with block-cyclic distributions were not considered. KALI does not support the data alignment phase and hence the problem of local memory allocation with hole compression does not arise.

Compilation of array statements in Distributed Fortran 90 is described in [92], but Distributed Fortran 90 supports only the block distribution. Furthermore, alignments are not included in the language definition.

For an array expression of the form $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ with the arrays distributed using block-cyclic distributions, Paalvast et al. [96, 97] present techniques to enumerate the portion of $B$ which is modified by a processor $p$ and the portions of array $A$ which reside on $p$ but are needed by the other processors. Their communication scheme is based on scanning over the active array indices of array sections to determine the elements which need to be communicated. This scheme will incur a high run-time overhead since a local-to-global and a global-to-local translation will be needed for each active element in order to determine the destination processor. Alignments are not considered and hence no explicit method for performing memory allocation was presented. Recent concurrent work by Reeuwijk et al. develops a unified method for the derivation of two forms for both local index set enumeration and local storage assignment [128] for arrays distributed using a two-level mapping. The two forms proposed have similarities with the virtual processor approach - one corresponding to the virtual-block view and the other to the virtual-cyclic view. However, the approach in [128] does not attempt a closed form characterization of the processor-
sets with respect to each source/destination processor. The method for eliminating holes while allocating memory has similarities to that proposed in Chapter IV.

The problem of active index-set identification for array statements involving block-cyclically distributed arrays was addressed by Chatterjee et al. [30, 31] using a finite-state machine (FSM) to traverse the local index space of each processor. If all arrays in an array statement have the same block-cyclic distribution and access stride, the order of access of the local elements with the FSM approach turns out to be the same as the access order when the virtual-block view is taken with our approach. However, if the distribution or access stride of the array section on the left-hand-side is different from that of an array section on the right-hand-side, it appears that after determination of the active local indices of the right-hand-side section using an FSM, an explicit local-to-global translation corresponding to the right-hand-side section and a global-to-local translation corresponding to the left-hand-side section will need to be performed for each active element. In [30, 31], restricted cases involving arrays with different strides are treated, but even with these, the generation of communication sets requires explicit index translation for each active element. With the virtual processor approach explicit local-to-global and global-to-local translation is not needed, even when the array sections differ in both distribution and access stride.

Recent independent work by Stichnoth also addresses the problem of active index-set and communication-set identification for array statements involving block-cyclically distributed arrays [117, 118]. The formulation proposed has similarities to the virtualization scheme with a virtual-cyclic view at both the source and target array.
However, the approach in [117, 118] does not attempt a closed form characterization of the active processor-sets with respect to each source/destination processor. Although the proposed scheme does not require the inclusion of indexing data in the message, a protocol in which a processor sends both the data values and the indices on the receiving processor, to facilitate unpacking, is proposed and evaluated. This strategy increases the data volume transmitted but improves performance on machines with high communication bandwidth such as the iWarp system. No explicit formulation for handling two-level mappings is provided in the proposed methods.

The Syracuse F90-D compiler's initial implementation [20, 19, 21] uses compile-time characterization of communication only for block distributions [2] and relies upon run-time generation of schedules for the general block-cyclic case using the approach adopted by the PARTI system [113]. The proposed methods do not try to eliminate holes introduced due to an alignment with non-unit stride.

The implementation of the Fortran-D compiler at Rice University initially focused on block distributions [64, 65, 66, 125]. Extensions to handle arrays with block-cyclic distributions are developed in [62, 63]. Their approach for determining communication sets is based upon translating the set of local indices accessed by the left-hand-side and right-hand-side reference to global indices and computing the intersection of these data index sets. The data index sets in global indices are translated so that the first elements match and the intersection is determined using a scanning approach which efficiently exploits the repetitive pattern of the intersection of the two index sets. An approach similar to the FSM approach [30] for determining the local mem-
ory access sequence is used. Efficient techniques for the FSM table generation are presented for special cases. A linear-time algorithm for constructing the FSM which improves the asymptotic complexity of the algorithm presented in [30] was developed in [78]. The key idea behind the construction is the recognition that the local index space for a block-cyclically distributed array is an integer lattice spanned by a basis of dimensionality two. Extensions of the FSM approach to handle arbitrary affine subscripts for nested loops are presented in [77]. The approach relies upon run-time methods based on the inspector-executor paradigm [113] to handle the general case.

An approach for the local index set and communication set enumeration based on identifying the basis for the integer lattice and exploiting repetition in the send and receive sets was presented in [124].

The use of the virtual processor approach for addressing the problem of active index-set identification for array statements involving block-cyclic distributions and that of redistribution of arrays with block-cyclic distributions was first presented in [58, 59]. Explicit closed-form characterization of processor sets using regular sections, development of a cost model to determine the virtualization scheme with lowest indexing time, and an experimental verification of the efficacy of the approach were reported in [57]. Typically, several array statements in HPF programs involve arrays which are identically distributed and array sections which are similar to those in previously executed statements. Techniques for identifying situations under which the previously evaluated sets can be reused, or efficient methods used for incremental evaluation of the index sets were developed in [76].
Recently, several methods based on expressing the problem of enumerating the local, processor, and data index sets as a "scanning polyhedra" problem have been presented. The "scanning polyhedra" problem takes as input a set of linear constraints in the form of equalities and inequalities among variables. The set of linear constraints represents a convex polyhedron. The output is a nested loop that scans all the integer points in the convex polyhedron in lexicographic order. Methods for solving the scanning polyhedra problem based on the Fourier-Motzkin pairwise elimination method [114, 136] have been presented in [6, 136]. This method has been used for generating code for distributed-memory machines starting from a sequential program [4, 5, 136]. Techniques which use the scanning polyhedra method for generating the index sets have been presented in [7, 48, 49, 126].

The method proposed in [126], requires an additional loop in the loop nest for enumerating the data send and receive sets, than that required by the virtual processor approach. Furthermore, the generated loops bounds and local array subscripts are complicated. While no direct comparison with the virtual processor approach has been performed, the existence of the additional loop and the cost of evaluating complicated loop bounds for each iteration is expected to lead to a greater indexing cost. Furthermore, while data alignments can be incorporated, the methods proposed in [126] do not consider data alignments.

The method based on the scanning polyhedra approach proposed in [7], converts the system of linear equalities into the Hermite Normal Form before applying the scanning polyhedra approach. This conversion reduces the number of variables in
the system of equalities and the number of loops in the generated loop nest is equal to that required by the virtual processor approach. However, the generated loop bounds are complicated. The conversion of the set of equalities into the Hermite Normal Form essentially constructs the basis for scanning the index sets which form an integer lattice. The memory allocation problem is addressed by allocating the local array as a two-dimensional array and compressing the holes along each basis vector of the lattice. This method has similarities with that used for hole compression in the virtual processor approach.

The techniques based on the scanning polyhedra approach presented in [48, 49] focus only on block distributions. Since two-level mappings are not considered, no solution to the memory allocation problem is provided.

A problem with all the schemes based on the scanning polyhedra approach is that several decisions have to be made at compile-time even though sufficient information may not be available. For instance, while generating the loop nest, ordering the loops such that the innermost loop has the largest size and the outer loops have smaller size, leads to lower indexing costs. The heuristics developed for selecting the appropriate virtualization scheme in the virtual processor approach are based on this rule of thumb. If sufficient information in not available at compile-time, then the methods based on the scanning polyhedra approach have to default to a particular order in the loop nest which may not be the best order. For the virtual processor approach, the choice of the virtualization scheme can be made at run-time which makes it a more flexible and efficient scheme for the execution of the array statement.
1.3.2 Array Redistribution

Several methods for performing the indexing for array redistribution have been presented in the literature. Closed form expressions for determining the processor and data index sets for redistributing block and cyclically distributed arrays were developed in [58, 82]. A virtual processor approach for performing redistributions involving block-cyclically distributed arrays was proposed in [58]. The virtual processor approach requires some additional memory and computation at runtime to efficiently enumerate the processor and data index sets. Efficient runtime strategies for performing redistribution for special cases involving block-cyclically distributed arrays were presented in [123]. The developed strategies consider array redistributions in which the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. For the general redistribution problem involving block-cyclically distributed arrays, an approach which incurs a significant indexing overhead per array element was needed. An array redistribution algorithm which can handle arbitrary source and target processor sets and block-cyclic distributions was presented in [105, 106]. Similar to the virtual processor approach presented in [58], this approach would, in general, incur an additional runtime overhead for solving linear diophantine equations. We developed closed form expressions for the processor and data index sets for some special cases of array redistribution involving block-cyclically distributed arrays in [73, 74, 75]. The solution to the general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. The developed closed forms provide
a precise representation for the index sets scanned using the algorithms presented in [123].

Several methods for performing the complete exchange - a special case of the all-to-many personalized communication in which each processor communicates with every other processor - have been presented in the literature [15, 16, 17, 18, 42, 55, 72, 115, 120, 121]. A multi-phase approach for the complete exchange on hypercube-connected multicomputers was presented in [16]. The underlying idea of the multi-phase approach is to combine data to be sent to a set of processors into one message and indirectly send it through this set of processors - each processor appropriates the segment destined to it from this message, repackages the message along with its local data for one of the destination processors and sends it to the next processor in the set. The use of the combining strategy reduces the number of message startups required for the complete exchange but incurs an additional data transmission cost as a data element has to be packed, transmitted, and unpacked multiple times. Similar combining schemes for two-dimensional meshes were presented in [18, 55, 120, 121].

The issue of scheduling the all-to-many personalized communication for array redistribution has received relatively little attention. Static and run-time scheduling techniques to reduce node and link contention in performing general irregular all-to-many personalized communication were presented in [107, 109, 108, 110, 131]. In the runtime techniques presented therein, each processor constructs a global communication matrix and replicates the computation for scheduling the communication using this matrix.
In the context of reducing communication cost for array redistribution, strip-mining redistribution, a method in which portions of the array are redistributed in sequence, as opposed to redistributing the entire array in one step to obtain good communication-computation overlap, was presented in [130, 129]. In the context of modeling link contention, a static approach for estimating network contention on an e-cube routed hypercube for parallel programs generated in the Vienna Fortran Compiling System [25, 28, 29] for block distributed arrays was presented in [45, 44]. Conditions were developed for identifying situations under which network contention on e-cube routed hypercubes would arise and a class of statements for which no contention will occur was identified.

Few methods have exploited the existence of patterns in the all-to-many personalized communication for array redistribution, or developed a multi-phase scheduling scheme to reduce the communication cost for array redistribution. Techniques for scheduling the communication for array redistribution which utilize the fact that the all-to-many communication arises from special forms of array redistribution, were developed in [73]. A communication cost model for redistribution and a multi-phase approach to reduce the communication cost were also developed. The developed techniques were applicable when the source and the target block-cyclic distributions satisfied certain conditions. Extensions to handle arbitrary source and target block-cyclic distributions were presented in [74, 75].
1.4 Organization of Thesis

The organization of the thesis is as follows. Chapter II describes the constructs provided in HPF-like languages for mapping arrays to the processors of a distributed-memory machine. In Chapter III, techniques for efficient execution of an array statement involving arrays distributed using the one-level mappings provided in HPF-like languages are developed. Extensions of the techniques for array statement execution to arrays distributed using two-level mappings are described in Chapter IV. Array redistribution corresponds to a special form of the array statement. Closed form expressions for the index sets for performing array redistribution are developed in Chapter V. In Chapter VI, a communication cost model for array redistribution is developed and multi-phase redistribution, a method for reducing the communication cost of array redistribution is presented. Conclusions and directions for future work are provided in Chapter VII.
CHAPTER II

Array Statement Execution on Distributed-Memory Machines

Languages such as High Performance Fortran (HPF) [46, 84, 86, 90], Fortran D [47], Vienna Fortran [27, 28, 29], PANDORE [43], and Distributed Fortran 90 [92] allow a programmer to annotate a program with alignment and distribution directives that describe the mapping of array elements to the processors of a distributed-memory machine. When compiling a program written in such a language, it is the responsibility of the compiler to partition the arrays among the processors and to produce node code with the required communication commands to transfer data between the local memories' of the processors at appropriate points in the execution of the parallel program.

The array assignment statement in Fortran 90 [1, 68, 93, 95] is provided for expressing simultaneous assignments to large groups of array elements. The multiple assignment functionality provides a convenient syntax for expressing data parallelism. The array assignment statement is very similar to the FORALL construct provided in HPF and the WHERE construct in Fortran 90. Methods developed for efficient execution of the array assignment statement involving distributed arrays, can be extended
in a straightforward fashion to a large class of statements expressed using the \texttt{FORALL} construct in HPF.

In this chapter, we first describe the alignment and distribution directives for mapping arrays onto processors. Since HPF has been accepted as a standard and since it includes the major language features required in this thesis, we use HPF syntax for expressing these directives. Next, we describe the syntax and semantics of the array assignment statement and address various issues related to the efficient execution of array statements involving arrays distributed among the processors of a distributed-memory machine.

2.1 Alignment and Distribution Directives

The HPF-like languages support a two-level mapping of data arrays to the processors of a distributed-memory machine. The language introduces an abstract Cartesian index domain referred to as a \textit{template}. Arrays are first \textit{aligned} to templates and templates are distributed across a programmer-defined grid of logical processors. The alignment of an array to a template and the subsequent distribution of the template determines the final mapping of the array. Fig. 1 illustrates the two-level mapping of data arrays onto the logical processors. In Fig. 1, a third level, which involves the mapping of the logical processors onto the physical processors of the underlying distributed-memory machine, is illustrated. This mapping is implementation specific and is considered out of the scope of the language definition. In this thesis, we shall use the terms physical processor and logical processor interchangeably. A segment of
Figure 1: The HPF two-level mapping model for data arrays.
an HPF program is shown in Fig. 2. The program segment defines a template $T$ of size 20 using the TEMPLATE directive. The alignment of arrays $A$ and $B$ with the template $T$ are defined using the ALIGN directive. Using the PROCESSORS directive, $PROC$ is defined to be a linear array of logical processors consisting of four processors. Finally the template is distributed onto the logical processors arrays using the DISTRIBUTE directive. Arrays $A$ and $B$ are effectively distributed among the processors of $PROC$. Note that the directives provide additional information to the compiler without affecting the semantics of the underlying program. We now describe the alignment and distribution directives in detail.

An array $A$ can be aligned with a template $T$ at a stride $c$ and an offset $a$, such that element $A(i)$ is mapped to template cell $T(ci+a)$. The alignment of the arrays $A$ and $B$ (Fig. 2) with the template $T$ are shown in Fig. 3. The alignment of array $A$ has an offset one and a stride two. The alignment of array $B$ has an offset zero and a unit
Figure 3: HPF array alignment illustration.

An alignment with an offset zero and a unit stride is referred to as the identity alignment. Two data elements are said to be aligned if they are mapped to the same template cell. In the program segment (Fig. 2), array elements $A(I)$ and $B(2*I + 1)$ are aligned. Alignment of two arrays guarantees that the aligned array elements are located on the same processor independent of the distribution of the template among the processors. For instance, in Fig. 2, the FOR loop will not require any communication, independent of the distribution of the template, since element $A(I)$ and element $B(2*I + 1)$ are aligned with the same template cell $T(2*I + 1)$. Multi-dimensional arrays are aligned with multi-dimensional templates by independently specifying the alignment of an array dimension with a template dimension. Arrays of a dimensionality lower than the template dimensionality can be embedded within the template or replicated.

The regular distributions supported in HPF are the block, the cyclic and the block-cyclic distribution. Consider a template $T(m : n)$ distributed onto $P$ processors. In a block distribution, contiguous blocks of size $\lceil \frac{n-m+1}{P} \rceil$ are allocated to the proces-
Figure 4: HPF array distribution illustration.
Table 1: Index mapping for regular data distributions. \( N = n - m + 1 \).

### Block Distribution

<table>
<thead>
<tr>
<th>domain</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>local to global</td>
<td>( i = \text{Glb}(i, p) = p(\lceil N/P \rceil) + l + m )</td>
</tr>
<tr>
<td>global to local</td>
<td>( l = \text{Loc}(i, p) = (i - m) \text{ mod } [N/P] )</td>
</tr>
<tr>
<td>global to proc</td>
<td>( p = \text{Proc}(i) = (i - m) \text{ div } [N/P] )</td>
</tr>
</tbody>
</table>

### Cyclic Distribution

<table>
<thead>
<tr>
<th>domain</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>local to global</td>
<td>( i = \text{Glb}(i, p) = lP + p + m )</td>
</tr>
<tr>
<td>global to local</td>
<td>( l = \text{Loc}(i, p) = (i - m) \text{ div } P )</td>
</tr>
<tr>
<td>global to proc</td>
<td>( p = \text{Proc}(i) = (i - m) \text{ mod } P )</td>
</tr>
</tbody>
</table>

### Block-cyclic Distribution

<table>
<thead>
<tr>
<th>domain</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>local to global</td>
<td>( i = \text{Glb}(i, p) = (l \text{ div } b)bP + bp + l \text{ mod } b + m )</td>
</tr>
<tr>
<td>global to local</td>
<td>( l = \text{Loc}(i, p) = ((i - m) \text{ div } P)b + (i - m) \text{ mod } b )</td>
</tr>
<tr>
<td>global to proc</td>
<td>( p = \text{Proc}(i) = ((i - m) \text{ div } b) \text{ mod } P )</td>
</tr>
</tbody>
</table>

\( i \): global index, \( m \leq i \leq n \),

\( l \): local index, \( 0 \leq l < b \ast \left( \lceil \frac{N}{P} \rceil \right) \),

\( p \): processor index, \( 0 \leq p < P \).
sors. In a *cyclic* distribution, template elements are assigned to the processors in a cyclic fashion. In a *block-cyclic* distribution with block size $b$, referred to as $cyclic(b)$, blocks of the template of size $b$ are allocated to the processors in a cyclic fashion. The block-cyclic distribution is the most general regular data distribution supported by the HPF-like languages. A block distribution corresponds to a $cyclic(\left\lceil \frac{n-m+1}{p} \right\rceil)$ distribution; a cyclic distribution corresponds to a $cyclic(1)$ distribution. In many algorithms the best data distributions are block-cyclic distributions since they represent a good compromise between reducing communication requirements and achieving satisfactory computational load balance. Dongarra et al. [41] have demonstrated the importance of block-cyclic distributions for efficient dense matrix algorithms on distributed-memory machines. Fig. 4 illustrates the distributions for an array of size 20 on four processors.

Due to the distribution of the template, the aligned array $A$ is divided into $P$ arrays, each residing in the local memory of one of the processors. The portion of array $A$ located in a processor's local memory is referred to as $A_{loc}$ in the SPMD node program generated by an HPF compiler. An element $A(i)$ of array $A$ has a *global* and a *local* index. The global index $i$ is the index in array $A$, as referenced in an HPF program. The index of element $A(i)$ in $A_{loc}$ is its local index. The relationships between the global index, the local index and the processor index for regular data distributions for an array aligned using an identity alignment are shown in Table 1. A processor is said to *own* those elements of the distributed array which are allocated to it.
The introduction of a stride in the array-template alignment, introduces "holes" in the array $A_{loc}$ of each processor as some template cells do not have any array elements mapped to them. Consider an array $A$ aligned with a template at a stride three. The template is distributed using a cyclic(4) distribution onto four processors. Fig. 5(a) illustrates the partition of the template cells after distribution among the processors. Only those template cells which have indices that are multiples of three have an array element aligned with them. Fig. 5(b) illustrates the memory allocation for $A_{loc}$ in which memory is allocated for all template cells independent of whether array elements aligned with them or not. Ideally, a compiler should not allocate memory in $A_{loc}$ for those template cells with which no array elements are aligned. Fig. 5(c) shows the local memory allocation for $A_{loc}$ with compression of the holes. While this memory optimization reduces the amount of memory wasted, it compounds the complexity of performing the indexing into the array $A_{loc}$ as is explained in Chapter IV.

Multi-dimensional templates are distributed on the logical processor array by independently specifying the distribution of each dimension onto the processor array. We now describe the syntax and semantics of the array assignment statement.

### 2.2 Array Assignment Statements

The array statement is included in Fortran 90 [1, 68, 93, 95], to provide a convenient syntax for simultaneous assignment to large groups of array elements. Such array assignments are the core of the data parallel computations targeted by the HPF-like
Figure 5: Memory allocation for array mapped using a two-level mapping with stride (a) Layout of template cells among processors. (b) Local array without hole-compaction. (c) Local array with hole compaction.
languages. The **FORALL** statement in HPF provides a similar multiple assignment functionality.

Consider two arrays \( A(m_1 : n_1) \) and \( B(m_2 : n_2) \) distributed on \( P_1 \) and \( P_2 \) processors, respectively. A simple form of the array assignment statement is:

\[
B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1)).
\]  

The array section \( A(l_1 : u_1 : s_1) \) consists of elements of \( A \) with indices \( \{l_1 + i * s_1 | 0 \leq i \leq [(u_1 - l_1)/s_1]\} \). The set of integers \( (l : u : s) \) is referred to as a *regular section*. In the array statement, \( B(l_2 + i * s_2) \) is assigned the value \( f(A(l_1 + i * s_1)) \). \( A(l_1 + i * s_1) \) and \( B(l_2 + i * s_2) \) are referred to as corresponding elements in the array assignment statement. The assignment can be executed independently for each pair of corresponding elements. The semantics of the array assignment require that the entire section \( A(l_1 : u_1 : s_1) \) is read, the result of \( f \) on each element of the section evaluated and written to the corresponding element of \( B(l_2 : u_2 : s_2) \). Semantically, this corresponds to copying \( A(l_1 : u_1 : s_1) \) to a temporary array section \( Tmp \), computing the function \( f \) on the elements of \( Tmp \) and assigning the resulting values to \( B(l_2 : u_2 : s_2) \). The evaluation of the array assignment \( B(0 : 12 : 2) = f(A(0 : 18 : 3)) \) for arrays \( A(0 : 23) \) and \( B(0 : 19) \), is illustrated in Fig. 6.

The owner-computes rule has been proposed as a compilation methodology for partitioning the computation among the processors \([24, 101, 111, 112, 138]\) and is prevalent among current compilers for distributed-memory machines. With the owner-computes rule the processor that owns an element of \( B(l_2 : u_2 : s_2) \) is responsible for performing the computation that modifies its value. Since \( A \) and \( B \) are distributed,
corresponding elements of $A(l_1 : u_1 : s_1)$ and $B(l_2 : u_2 : s_2)$ may not be located on the same processor. Some elements of $A(l_1 : u_1 : s_1)$ will be communicated to the processor owning the corresponding elements of $B(l_2 : u_2 : s_2)$. In the generated code, all the non-local data needed by a processor is fetched into temporary arrays in a processor’s local memory using interprocessor communication. In order to reduce the communication overhead, each processor first determines all the data it needs to send to and receive from other processors and then performs the needed communication. This reduces the communication overhead by aggregating all the data movements needed from one processor to another into a single message. After communication each processor then performs the computation in its local address space. To this purpose, each processor must determine the set of array elements in its local address space whose values it must compute. Fig. 7, illustrates the evaluation of the array statement $B(0 : 12 : 2) = f(A(0 : 18 : 3))$ for arrays $A(0 : 23)$ and $B(0 : 19)$ which are now distributed using a block distribution on four processors. Processor 1 must
send the value of $A(6)$ ($A_{\text{loc}}(0)$ on processor 1) to processor 0 and receive the value of $A(12)$ which is needed in the computation of the final value for $B(8)$ ($B_{\text{loc}}(3)$ on processor 1). Next, it must compute new values for $B_{\text{loc}}(1:3)$.

Determining the local array elements to send, receive and compute during the execution of the array assignment statement, involves a high computation cost, primarily due to the complexity of the indexing functions relating the global, local, and processor indices for regular data distributions. In the following sections, we describe two methods for performing the necessary indexing. The first method, referred to as \textit{array-scanning}, serves the purpose of illustrating the problems with a naive approach for performing the requisite indexing. The second method is based on developing and using closed-form expressions for the necessary index sets and is the method of choice for execution of the array statement.
2.3 Array Statement Execution Using Array Scanning

For the execution of the array assignment statement using the array-scanning approach, the node program for a processor $p$ scans the entire local index space for array $A$ to determine the array elements it needs to send to other processors. For each index $i$ in the local index space corresponding to $A.loc$, the global array index $g_1$ is determined. If the array element $A(g_1)$ belongs to the array section $A(l_1 : u_1 : s_1)$, then the global index $g_2$, of the element corresponding to $A(g_1)$ in the section $B(l_2 : u_2 : s_2)$ is determined, the index $q$ of the processor on which $B(g_2)$ is located is determined, and $A.loc(i)$ is inserted into a buffer destined for processor $q$. After all the local array elements are scanned and the elements inserted into the appropriate buffers, the processor $p$ sends the buffers to the corresponding processors. Note that every buffer has to be communicated even if it is empty since a receiving processor does not have any information about the number of messages it will receive. A processor then receives messages from all the other processors. During the computation phase, the local index space corresponding to $B.loc$ is scanned. For each local index $i$, the corresponding global index $g_2$ is determined. If the element $B(g_2)$ belongs to the section $B(l_2 : u_2 : s_2)$, then the processor $q$ on which the element, corresponding to $B(g_2)$ in $A(l_1 : u_1 : s_1)$, is located, is determined and the value for that element obtained from the appropriate message buffer. The node code for the array-scanning approach is illustrated in Fig. 8. All code generated by the compiler is written using a Pascal-like syntax. The functions $Glb.A$ and $Glb.B$ correspond to the local-to-global index conversion functions for the data distributions of arrays $A$ and $B$, respectively
while the functions $Proc_A$ and $Proc_B$ are corresponding global-to-processor index
conversion functions.

The array-scanning approach requires a local-to-global and a global-to-processor
conversion for every index in $A_{loc}$ and $B_{loc}$. Thus the indexing overhead is signifi-
cantly high and is independent of the amount of computation involved in the array
assignment statement. The indexing overhead is greater when the array is mapped
using a two-level mapping with a non-unit stride in the alignment and hole compres-
sion is performed while allocating memory for the local arrays. Furthermore this
approach has a large temporary memory requirement. The temporary space required
for sending and receiving messages would in the worst case be equal to the size of
the array $A_{loc}$ and $B_{loc}$, respectively. A third problem with the array scanning ap-
proach is that no overlap between the indexing computation and the communication
is possible, as all the indexing for the data to be sent has to be performed before any
data can be communicated and all the data has to be received before any compu-
tation for the array assignment can be performed. Such an overlap is desirable in a
message-passing program to hide the communication cost. In the following section,
we propose an approach for the execution of the array assignment statement which
alleviates the problems associated with the array-scanning approach. This method is
based on using closed form expressions for various index sets.
/* Node code for processor p */

/* Send phase */
cnt(0 : P_2 - 1) = 0;
for i = 0, sizeof(A.loc) - 1
    g_1 = Glb_A(i, p);
    if (l_1 \leq g_1 \leq u_1) \land ((g_1 - l_1) \mod s_1 == 0) then
        j = (g_1 - l_1) \div s_1; g_2 = l_2 + js_2; q = Proc_B(g_2);
        tmp.out(q, cnt(q)) = A.loc(i); cnt(q) = cnt(q) + 1;
    endif
endfor
for q = 0, P_2 - 1
    send(q, tmp.out(q), cnt(q));
endfor

/* Receive and execution phase */
for q = 0, P_1 - 1
    recv(q, tmp.in(q), sizeof(tmp.in(q)));
endfor
cnt(0 : P_2 - 1) = 0;
for i = 0, sizeof(B.loc) - 1
    g_2 = Glb_B(i, p);
    if (l_2 \leq g_2 \leq u_2) \land ((g_2 - l_2) \mod s_2 == 0) then
        j = (g_2 - l_2) \div s_2; g_1 = l_1 + js_1; q = Proc_A(g_1);
        B.loc(i) = f(tmp.in(q, cnt(q))); cnt(q) = cnt(q) + 1;
    endif
endfor

Figure 8: Array assignment execution using array scanning.
2.4 Array Statement Execution Using Index Sets

We first define the data and the processor index sets required for the execution of the array assignment statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$. For the array section $A(l_1 : u_1 : s_1)$, each processor $p$, $0 \leq p < P_1$ has to evaluate the following information for every other processor $q$, $0 \leq q < P_2$.

- $DSend(p,q)$: the set of local indices of the data elements that $p$ owns and are required by $q$. Processor $p$ has to send this set of elements to processor $q$.

Similarly, for the array section $B(l_2 : u_2 : s_2)$, each processor $p$, $0 \leq p < P_2$ has to evaluate the following information for every other processor $q$, $0 \leq q < P_1$.

- $DRecv(p,q)$: the set of local indices of the data elements that $p$ needs and are located on $q$. Processor $p$ will receive this set of data elements from $q$.

- $LIndex(p)$: the set of local indices of the elements of $B(l_2 : u_2 : s_2)$ that $p$ owns. Processor $p$ has to compute new values for this set of elements as would be evaluated by the array assignment statement.

For instance, in Fig. 7, the local index set for processor 0 corresponding to the array section $B(0 : 12 : 2)$ is $LIndex(0) = (0 : 4 : 2)$, while that for processor 1 is $LIndex(1) = (1 : 4 : 2)$. Processor 0 has to receive data corresponding to the global section $A(6 : 6)$ from processor 1. Thus, $DSend(1,0) = (0 : 0)$ and symmetrically $DRecv(0,1) = (4 : 4)$. Note that $DSend$ and $DRecv$ are expressed in terms of the local indices on the corresponding processors and in general
\( DSend(p, q) \neq DSend(q, p) \). The sets \( DSend(p, q) \) and \( DRecv(p, q) \) for a processor \( p \), are defined for all processors \( q, \, 0 \leq q < P_2 \) and could be empty for some processor \( q \). Enumerating the data index sets will involve some additional computational overhead even if the sets eventually turn out to be empty. Thus a scheme which would iterate over all the processors and perform the computation for enumerating the send and receive data index sets would in general perform a significant amount of redundant computation. Ideally the additional computation should be performed only for enumerating the data send and receive sets which are not empty. Hence, two additional sets of processor indices are defined as follows.

- \( PSend(p) \): is the set of processors to which processor \( p \) has to send some array elements, i.e., \( DSend(p, q) \) is not empty. Thus \( PSend(p) = \{ q \mid DSend(p, q) \neq \emptyset, \, 0 \leq q < P_2 \}, \, 0 \leq p < P_1 \).  

- \( PRecv(p) \): is the set of processors from which processor \( p \) has to receive some array elements, i.e., \( DRecv(p, q) \) is not empty. Thus \( PRecv(p) = \{ q \mid DRecv(p, q) \neq \emptyset, \, 0 \leq q < P_1 \}, \, 0 \leq p < P_2 \).

Using these sets, the node program pseudo-code for the execution of the array statement is as shown in Fig. 9. An optimization that is possible if the array assignment statement has a single array section on the right hand side is to combine the receiving phase with the execution phase. Rather than unpacking a message into a local temporary location and separately evaluating the function \( f \) on each element of the local section, \( f \) is applied on each element directly from the message buffer. This optimization reduces the overhead of an additional loop to traverse the local elements
/* Node code for processor p */

/* Send phase */
for q ∈ PSend(p)
    Tmp(0 : |DSend(p,q)| – 1) = Aloc(DSend(p,q));
    send(q,Tmp,|DSend(p,q)|);
endfor

/* Receive phase */
for q ∈ PRecv(p)
    recv(q,Tmp,|DRecv(p,q)|);
    T(DRecv(p,q)) = Tmp(0 : |DRecv(p,q)| – 1);
endfor

/* Execution Phase */
for i ∈ LIIndex(p)
    Bloc(i) = f(T(i))
endfor

Figure 9: Array statement execution using index sets.

of the array sections and allows communication-computation overlap which is useful to hide communication latency. However, this optimization may not be applicable when the array expression has multiple array sections on the right hand side.

Thus the efficient enumeration of data and processor send and receive sets and the local index sets is important for the execution of array assignment statements. For block and cyclic distributions, the send and receive data index and processor sets can be expressed as regular sections. However, for block-cyclic distributions, these sets cannot be expressed as simple regular sections. For instance, consider the local index set for the array section A(0 : 39 : 2) when A(0 : 39) is distributed onto
four processors using a cyclic(5) distribution, as shown in Fig. 10. The array section elements are separated by a constant stride 2 in global index space. However, the elements of the array section do not have a constant stride in the local index space of a processor. For example, $LIndex(0) = (0, 2, 4, 5, 7, 9)$ and $LIndex(1) = (1, 3, 6, 8)$.

Similarly consider the array section $A(0 : 36 : 2)$ of the array $A(0 : 37)$ shown in Fig. 5. The elements belonging to the array section are marked in Fig. 11. In global address space, the array section elements are separated by a constant stride 4. However, in local address space the elements do not have a constant stride. For example, without “hole-compression”, as seen in Fig. 11(b), $LIndex(0) = \{0, 6, 22\}$ while $LIndex(2) = \{1, 7, 16, 22\}$. A similar situation arises when compression of holes in the local array is performed while allocating memory for the local array. With hole-compression the additional problem of partitioning the array and allocating memory only for the template cells which have array elements aligned with them has to be addressed.

The code for statements involving multiple right hand side array sections is generated by repeating the simple array assignment. Let $T_1$ and $T_2$ be temporary arrays.

![Figure 10: Variable stride in $A.loc$ for global array section $A(0 : 40 : 2)$.

<table>
<thead>
<tr>
<th>Procs :</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Indices</td>
<td>1 3</td>
<td>5 7</td>
<td>10 12</td>
<td>15 17</td>
</tr>
<tr>
<td>Local Indices</td>
<td>21 23</td>
<td>25 27</td>
<td>30 32</td>
<td>35 37</td>
</tr>
<tr>
<td></td>
<td>1 3</td>
<td>0 2</td>
<td>0 2</td>
<td>2 4</td>
</tr>
<tr>
<td></td>
<td>6 8</td>
<td>5 7</td>
<td>5 7</td>
<td>7 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 9</td>
<td>9 4</td>
<td>4 9</td>
</tr>
</tbody>
</table>
Figure 11: Memory allocation for an array mapped using a two-level mapping with stride.
\[
\begin{align*}
S_1 : \quad T_1(l_2 : u_2 : s_2) &= A(l_1 : u_1 : s_1) \\
S_2 : \quad T_2(l_2 : u_2 : s_2) &= C(l_3 : u_3 : s_3) \\
S_3 : \quad B(l_2 : u_2 : s_2) &= T_1(l_2 : u_2 : s_2) + T_2(l_2 : u_2 : s_2).
\end{align*}
\]

Figure 12: Array statement execution involving multiple right hand side arguments.

with identical alignment and distribution as \( A \). The evaluation of \( B(l_2 : u_2 : s_2) = A(l_1 : u_1 : s_1) + C(l_3 : u_3 : s_3) \) is performed as shown in Fig. 12. The code for \( S_1 \) and \( S_2 \) can be generated as shown in Fig. 9. Since \( T_1 \) and \( T_2 \) are identically distributed and aligned as \( A \), execution of statement \( S_3 \) will not require any communication and only the execution phase will suffice.

2.5 Summary

In this chapter, we described the alignment and distribution directives for mapping arrays onto processors of a distributed-memory machine. We then presented the syntax and semantics of the array assignment statement. A naive method for execution of the array assignment statement which incurs a high indexing cost was presented. A method based on closed form expressions for the processor and data index sets was proposed for efficient execution of the array assignment statement involving distributed arrays.

In Chapter III, we develop the closed form expressions for the processor and data index sets for arrays distributed using block or cyclic distributions. We then propose a virtual processor approach for efficiently enumerating the index sets for
block-cyclically distributed arrays. In Chapter IV, we describe a methodology for performing local memory allocation and a strategy for extending the methods developed for one-level mappings to those for two-level mappings with non-unit strides.
CHAPTER III

Index Sets for One-Level Mappings

In this chapter, we develop closed form expressions for the local index sets and the communication sets for the execution of array statements involving arrays distributed using identity mappings, i.e., the array is aligned with the template at a unit stride with offset zero. Thus each element of the template is associated with an element of the array and after distribution the local array does not contain “holes”. The chapter is organized as follows. In Section 3.1, we develop closed form expressions for the local index sets for block and cyclically distributed arrays. We next develop closed form expressions for the processor and data index sets for block and cyclically distributed arrays in Section 3.2. The virtual processor approach for handling block-cyclically distributed arrays is then presented in Section 3.3. This approach is based on viewing a block-cyclic distribution as a block (or cyclic) distribution on a set of virtual processors, which are cyclically (or block-wise) mapped to the physical processors. These views are referred to as a virtual-block or virtual-cyclic views depending on whether a block or a cyclic distribution of the array on the virtual processors is used. The virtual processor approach permits different schemes based on the combination of the virtual processor views chosen for the different arrays involved in an array statement.
These virtualization schemes have different indexing overhead. We present a strategy for identifying the virtualization scheme which will have the best performance. Extensions of the indexing strategies for executing array statements involving multidimensional arrays are presented in Section 3.4. Performance results on a 32-node Cray T3D system for hand-compiled code for array assignments are presented in Section 3.5. The results show that using the virtual processor approach, efficient code can be generated for execution of array statements involving block-cyclically distributed arrays. Finally, some discussion concerning the extension of the developed schemes to handle different forms of data parallel statements and some of the limitations are provided in Section 3.6.

3.1 Local Index Sets for the Array Statement

Consider an array \( B(m_2 : n_2) \) distributed onto \( P_2 \) processors using a block or cyclic data distribution and the array section \( B(l_2 : u_2 : s_2) \). We now develop closed form expressions for the local index sets. We first consider the case when the stride \( s_2 > 0 \) and then describe modifications for negative strides.

3.1.1 Block Distribution

Let array \( B \) be block distributed over \( P_2 \) processors. The block size of array \( B \) is \( b_2 = \lceil \frac{n_2 - m_2 + 1}{P_2} \rceil \). Consider a processor \( p \) which accesses the elements of the array section. The global indices of the array elements allocated to \( p \) are given by the regular section \( (m_2 + p b_2 : \min(m_2 + p b_2 + b_2 - 1, n_2)) \). Let \( B(l_2 + i_2 s_2) \) and \( B(l_2 + j_2 s_2) \) be the first and last array elements of \( B(l_2 : u_2 : s_2) \) allocated to \( p \). Thus \( i_2 \) will be the smallest
positive value for $i$ on processor $p$ such that $l_2 + is_2 \geq m_2 + pb_2$. Similarly, $j_2$ will be the largest positive value for $j$ on processor $p$ such that $l_2 + j_2s_2 \leq \min(m_2 + pb_2 + b_2 - 1, n_2)$.

Simplifying the above inequalities, we obtain $i_2$ and $j_2$ as follows:

$$i_2 = \left\lfloor \frac{\max(m_2 + pb_2 - l_2, 0)}{s_2} \right\rfloor,$$

$$j_2 = \left\lfloor \frac{\min(m_2 + pb_2 + b_2 - 1 - l_2, u_2 - l_2)}{s_2} \right\rfloor. \tag{3.1} \tag{3.2}$$

Note that, if $i_2 > j_2$, then no elements of the array section are located on processor $p$. The stride between consecutive elements of the array section in the local index space of the processor is $s_2$. Translating the global indices of the first and the last slice indices to the local indices on processor $p$, as illustrated in Fig. 13, it follows that the local index set on processor $p$ is the regular section given by:

$$LIndex(p) = (l_2 + i_2s_2 - pb_2 : l_2 + j_2s_2 - pb_2 : s_2) \tag{3.3}$$

We now construct the local index set for a cyclically distributed array.
3.1.2 Cyclic Distribution

Let array $B$ be cyclically distributed over $P_2$ processors. Consider a processor $p$. The global indices of the set of array elements located on processor $p$ is given by the regular section $(p + m_2 : n_2 : P)$. Let $B(l_2 + i_2 s_2)$ and $B(l_2 + j_2 s_2)$ be the first and last array elements of the array section $B(l_2 : u_2 : s_2)$ allocated to $p$. Processor $p$ may contain elements of the array section $B(l_2 : u_2 : s_2)$ if and only if the linear Diophantine equation:

$$l_2 + i s_2 = m_2 + p + cP_2 \quad (3.4)$$

in integer variables $i$ and $c$, has a solution. The linear Diophantine equation has a solution if and only if $gcd(s_2, P_2)|(m_2 + p - l_2)$, i.e., $m_2 + p - l_2$ is divisible by $gcd(s_2, P_2)$ [23]. The general solution to the linear Diophantine equation $i s_2 - cP_2 = k$ is found as follows. Let $g_2 = gcd(s_2, P_2) = \alpha s_2 + \beta P_2$, where $\alpha, \beta \in \mathbb{Z}$. The quantities $\alpha$ and $\beta$ can be determined using the extended Euclid algorithm [81]. Then the general solution for the equation is the parameterized family with parameter $t \in \mathbb{Z}$ given by:

$$i = \frac{k \alpha}{g_2} + \frac{P_2}{g_2} t, \quad c = -\frac{k \beta}{g_2} + \frac{s_2}{g_2} t \quad (3.5)$$

Using the parameterized solutions in Eq. 3.5, the solution pair $(i_2, c_2)$ can be found such that $i_2$ is the smallest non-negative integer for which the corresponding $c_2$ is also a non-negative integer. The slice index of the first array section element located on processor $p$ is $i_2$. Taking these constraints into account and eliminating the parameter
\[ i_2 = \frac{(m_2 + p - l_2)\alpha}{g_2} + \frac{P_2}{g_2} \left( \left\lceil \frac{-(m_2 + p - l_2)\alpha}{g_2} \right\rceil \right) \]  

(3.6)

The slice index \( j_2 \) of the last array section element located on processor \( p \) is found from \( i_2 \) as

\[ j_2 = i_2 + \left\lceil \left( \frac{u_2 - l_2 - i_2 s_2}{\text{lcm}(s_2, P_2)} \right) \right\rceil \cdot \text{lcm}(s_2, P_2) \]  

(3.7)

Now, if \((i_2, c_2)\) is a solution pair for the set of equations Eq. 3.5, then \((i_2 + \frac{P_2}{g_2}, c_2 + \frac{s_2}{g_2})\) is also a solution pair. Since \(c_2\) corresponds to the local index on processor \( p \) of the slice \( j_2 \), the stride between elements of the array section in the local index space is \( \frac{s_2}{g_2} \). Thus the local index set on processor \( p \) is given by the regular section:

\[ L\text{Index}(p) = \left( \left\lfloor \frac{l_2 + i_2 s_2}{P_2} \right\rfloor \cdot \text{div} P_2 : \left\lfloor \frac{l_2 + j_2 s_2}{\frac{s_2}{g_2}} \right\rfloor \cdot \text{div} P_2 : \frac{s_2}{g_2} \right) \]  

(3.8)

### 3.1.3 Modifications for Negative Strides

The modifications to the local index sets if the array section \( B(l_2 : u_2 : s_2) \) has a stride \( s_2 < 0 \) are straightforward. The array section \( B(l_2 : u_2 : s_2) \) with a stride \( s_2 < 0 \), is equivalent to the array section \( B(u'_2 : l_2 : -s_2) \), where \( u'_2 = u_2 + (l_2 - u_2) \mod s_2 \). Since the array section \( B(u'_2 : l_2 : -s_2) \) has a positive stride the parameterized closed forms developed in the previous sections are directly applicable. The construction of the new array section reverses the direction of the elements in the section. We now construct the communication sets for the array statement.
3.2 Communication Sets for the Array Statement

Consider two arrays $A(m_1 : n_1)$ and $B(m_2 : n_2)$ distributed using a block or cyclic data distribution and the array statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$. We now develop closed form expressions for the send and receive processor and data index sets. In the following sections, $((l : u : s) * c - d)$ is defined as the regular section $(l * c - d : u * c - d : s * c)$. Also $((l : u : s) \text{ op } c)$, where $\text{op} \in \{\text{mod, div}\}$, consists of the integers $\{(l + i * s) \text{ op } c | 0 \leq i \leq \lfloor \frac{u-1}{s} \rfloor\}$. We initially derive the closed forms for the cases when $s_1 > 0$ and $s_2 > 0$. We then describe modifications to handle negative strides.

3.2.1 Block Distribution to Block Distribution

Let $A$ and $B$ be block distributed over $P_1$ and $P_2$ processors, respectively. The block size of array $A$ is $b_1 = \left\lceil \frac{n_1 - m_1 + 1}{P_1} \right\rceil$ and the block size of array $B$ is $b_2 = \left\lceil \frac{n_2 - m_2 + 1}{P_2} \right\rceil$.

Let processor $p$ be the sending processor and $A(l_1 + i_1 s_1)$ and $A(l_1 + j_1 s_1)$ be the first and last array elements of $A(l_1 : u_1 : s_1)$ allocated to $p$. We obtain $i_1$ and $j_1$ as follows:

$$i_1 = \left\lceil \frac{\max(m_1 + pb_1 - l_1, 0)}{s_1} \right\rceil,$$

$$j_1 = \left\lfloor \frac{\min(m_1 + pb_1 + b_1 + 1 - l_1, u_1 - l_1)}{s_1} \right\rfloor,$$

where $m_1 + pb_1$ and $m_1 + pb_1 + b_1 - 1$ are the global indices of the first and last elements of array $A$ allocated to $p$. The send processor set of $p$ includes processors that own elements in $B(l_2 + i_1 s_2 : l_2 + j_1 s_2 : s_2)$. Note that the send processor set is empty if $i_1 > j_1$. 
The closed form of the send processor set must include a processor index exactly once. The first and last processors that \( p \) will send data to are \((l_2 + i_1 s_2 - m_2) \div b_2\) and \((l_2 + j_1 s_2 - m_2) \div b_2\), respectively. We consider two cases. If the stride \( s_2 \) is less than the block size \( b_2 \), \( p \) will send at least one element to processor \( q \), where \((l_2 + i_1 s_2 - m_2) \div b_2 \leq q \leq (l_2 + j_1 s_2 - m_2) \div b_2\). For instance, Fig. 14 illustrates an example in which \( s_2 < b_2 \). If \( s_2 \geq b_2 \), processor \( p \) will send exactly one element to each receiving processor. Thus we have

\[
PSend(p) = \begin{cases} 
(l_2 + i_1 s_2 - m_2) \div b_2 : (l_2 + j_1 s_2 - m_2) \div b_2 & \text{if } i_1 \leq j_1 \land s_2 < b_2, \\
((i_1 : j_1) \ast s_2 + l_2 - m_2) \div b_2 & \text{if } i_1 \leq j_1 \land s_2 \geq b_2, \\
\emptyset & \text{if } i_1 > j_1.
\end{cases}
\]

Let \( B(l_2 + i_2 s_2) \) and \( B(l_2 + j_2 s_2) \) be the first and last elements of \( B(l_2 : u_2 : s_2) \)
allocated to processor $q$. We obtain $i_2$ and $j_2$ as follows:

$$i_2 = \left\lceil \frac{\max(m_2 + qb_2 - l_2, 0)}{s_2} \right\rceil, \quad (3.12)$$

$$j_2 = \left\lfloor \frac{\min(m_2 + qb_2 + b_2 - 1 - l_2, u_2 - l_2)}{s_2} \right\rfloor. \quad (3.13)$$

The global indices of the first and last element sent from processor $p$ to processor $q$ are $l_1 + \max(i_1, i_2) \ast s_1$ and $l_1 + \min(j_1, j_2) \ast s_1$, respectively. Converting global indices to local indices on processor $p$ we have:

$$DSend(p, q) =$$

$$(l_1 + \max(i_1, i_2) \ast s_1 - m_1 - pb_1 : l_1 + \min(j_1, j_2) \ast s_1 - m_1 - pb_1 : s_1). \quad (3.14)$$

For the receiving phase of processor $p$, the receive processor and data index sets can be determined similarly. Let $i_1$ and $j_1$ be the first and last slice indices of $A$ on processor $q$:

$$i_1 = \left\lceil \frac{\max(m_1 + qb_1 - l_1, 0)}{s_1} \right\rceil, \quad (3.15)$$

$$j_1 = \left\lfloor \frac{\min(m_1 + qb_1 + b_1 - 1 - l_1, u_1 - l_1)}{s_1} \right\rfloor, \quad (3.16)$$

and $i_2$ and $j_2$ to be the first and last indices of $B$ on processor $p$:

$$i_2 = \left\lceil \frac{\max(m_2 + pb_2 - l_2, 0)}{s_2} \right\rceil, \quad (3.17)$$

$$j_2 = \left\lfloor \frac{\min(m_2 + pb_2 + b_2 - 1 - l_2, u_2 - l_2)}{s_2} \right\rfloor. \quad (3.18)$$

Then, the closed forms are as follows:

$$PRcv(p) = \begin{cases} ((l_1 + i_2s_1 - m_1) \div b_1 : (l_1 + j_2s_1 - m_1) \div b_1) & \text{if } i_2 \leq j_2 \land s_1 < b_1, \\ ((i_2 : j_2) \ast s_1 + l_1 - m_1) \div b_1 & \text{if } i_2 \leq j_2 \land s_1 \geq b_1, \\ \emptyset & \text{if } i_2 > j_2, \end{cases} \quad (3.19)$$
\[ DRecv(p, q) = (l_2 + \max(i_1, i_2) \cdot s_2 - m_2 - pb_2 : l_2 + \min(j_1, j_2) \cdot s_2 - m_2 - pb_2 : s_2). \] (3.20)

Node code for processor \( p \) developed using the closed forms is shown in Fig. 15 and Fig. 16. Fig. 15 illustrates the sending phase while Fig. 16 illustrates the receiving phase. Nonblocking sends and blocking receives are the communication primitives used. Fig. 15 and Fig. 16 show the most general form of the node code. It can be optimized by removing the unnecessary checks if some of the parameters are known at compile-time. In the sending phase, processor \( p \) uses the closed-form for \( DSend(p, q) \) to send messages to all processors \( q \in PSend(p) \). In the receiving phase, processor \( p \) first determines the number of messages it will receive using \( PRecv(p) \). Then, it receives the messages non-deterministically in the order they arrive. The receive command \( recv(q, tmp.buf) \) returns the sender processor's index in \( q \) and the message in \( tmp.buf \). The recv data index set \( DRecv(p, q) \) is used to unpack the received message. If \( p \in PSend(p) \), then the explicit message send and receive is replaced by a local memory copy of the buffers.

### 3.2.2 Block Distribution to Cyclic Distribution

Let \( A \) be block distributed and \( B \) be cyclically distributed over \( P_1 \) and \( P_2 \) processors, respectively. The block size of array \( A \) is \( b_1 = \left\lceil \frac{n_1 - m_1 + 1}{P_1} \right\rceil \) and the number of cycles for array \( B \) is \( C_2 = \left\lceil \frac{n_2 - m_2 + 1}{P_2} \right\rceil \). Let \( A(l_1 + i_1 s_1) \) and \( A(l_1 + j_1 s_1) \) be the first and last
/* Node Code for Processor p */

/* Sending Phase */

\[ i_1 = \max(m_1 + p \cdot b_1 - l_1, 0)/s_1 \]
\[ j_1 = \min(m_1 + p \cdot b_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1 \]

if \( i_1 \leq j_1 \wedge s_2 < b_2 \) then

for \( q = (l_2 + i_1 \cdot s_2 - m_2) \div b_2, (l_2 + j_1 \cdot s_2 - m_2) \div b_2 \)

\[ cnt = 0 \]
\[ i_2 = \max(m_2 + q \cdot b_2 - l_2, 0)/s_2 \]
\[ j_2 = \min(m_2 + q \cdot b_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2 \]
\[ l = l_1 + \max(i_1, i_2) \cdot s_1 - m_1 - p \cdot b_1 \]
\[ u = l_1 + \min(j_1, j_2) \cdot s_1 - m_1 - p \cdot b_1 \]

for \( r = l, u, s_1 \)

\[ tmp.buf.out(cnt) = A.loc(r) \]
\[ cnt = cnt + 1 \]

endfor

send\( (q, tmp.buf.out, cnt) \)

endfor

else if \( i_1 \leq j_1 \wedge s_2 \geq b_2 \) then

for \( k_1 = i_1, j_1 \)

\[ cnt = 0 \]
\[ q = (k_1 \cdot s_2 + l_2 - m_2) \div b_2 \]
\[ i_2 = \max(m_2 + q \cdot b_2 - l_2, 0)/s_2 \]
\[ j_2 = \min(m_2 + q \cdot b_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2 \]
\[ r = l_1 + \max(i_1, i_2) \cdot s_1 - m_1 - p \cdot b_1 \]
\[ tmp.buf.out(cnt) = A.loc(r) \]
\[ cnt = cnt + 1 \]

send\( (q, tmp.buf.out, cnt) \)

endfor

endif

Figure 15: Node code for \( B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1)) \) - Sending phase. \( A(m_1 : n_1) \) and \( B(m_2 : n_2) \) are block distributed on \( P_1 \) and \( P_2 \) processors, respectively.
/* Receiving Phase */

\[ i_2 = \frac{\max(m_2 + p \cdot b_2 - l_2, 0)}{s_2} \]

\[ j_2 = \frac{\min(m_2 + p \cdot b_2 + b_2 - 1 - l_2, u_2 - l_2)}{s_2} \]

if \( i_2 \leq j_2 \land s_1 < b_1 \) then

\[ \text{msg} = \frac{(l_1 + j_2 \cdot s_1 - m_1)}{b_1} \div \frac{(l_1 + i_2 \cdot s_1 - m_1)}{b_1} \]

else if \( i_2 \leq j_2 \land s_1 \geq b_1 \) then

\[ \text{msg} = j_2 - i_2 \]

else \( \text{msg} = -1 \)
endif

while \( \text{msg} \geq 0 \) DO

\text{recv}(g, \text{tmp} \_\text{buf} \_\text{in})

\[ \text{cnt} = 0 \]

\[ i_1 = \frac{\max(m_1 + q \cdot b_1 - l_1, 0)}{s_1} \]

\[ j_1 = \frac{\min(m_1 + q \cdot b_1 + b_1 - 1 - l_1, u_1 - l_1)}{s_1} \]

\[ l = l_2 + \max(i_1, j_2) \cdot s_2 - m_2 - p \cdot b_2 \]

\[ u = l_2 + \min(j_1, j_2) \cdot s_2 - m_2 - p \cdot b_2 \]

\text{DO} \ r = l, u, s_2

\[ B \_\text{loc}(r) = f(\text{tmp} \_\text{buf} \_\text{in}(\text{cnt})) \]

\[ \text{cnt} = \text{cnt} + 1 \]

\text{ENDDO}

\[ \text{msg} = \text{msg} - 1 \]

\text{ENDDO}

Figure 16: Node code for \( B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1)) \)- Receive Phase. \( A(m_1 : n_1) \)
and \( B(m_2 : n_2) \) are block distributed on \( P_1 \) and \( P_2 \) processors, respectively.
array elements of $A(l_1 : u_1 : s_1)$ located on processor $p$. We have

$$i_1 = \left\lfloor \frac{\max(m_1 + pb_1 - l_1, 0)}{s_1} \right\rfloor,$$  \hspace{1cm} (3.21)

$$j_1 = \left\lfloor \frac{\min(m_1 + pb_1 + b_1 - 1 - l_1, u_1 - l_1)}{s_1} \right\rfloor.$$

(3.22)

If $B(l_2 + i_1s_2)$ is located on processor $q$, then the next element of $B(l_2 : u_2 : s_2)$ located on $q$ is $B(l_2 + i_1s_2 + \text{lcm}(s_2, P_2))$. Furthermore, all the intermediate elements of $B(l_2 : u_2 : s_2)$, i.e., $B(l_2 + i_1s_2 : l_2 + i_1s_2 + \text{lcm}(s_2, P_2) - s_2 : s_2)$ must be located on distinct processors. Thus, the send processor set consists of processors owning elements $B(l_2 + i_1s_2 : l_2 + i_1s_2 + \text{lcm}(s_2, P_2) - s_2 : s_2)$, i.e.,

$$PSend(p) = (l_2 + (i_1 : \min(j_1, i_1 + \frac{\text{lcm}(s_2, P_2)}{s_2} - 1)) \cdot s_2 - m_2) \mod P_2 \hspace{1cm} (3.23)$$

For example, Fig. 17 illustrates an example in which $P_2 = 3$ and $s_2 = 2$. It can be seen that the stride distance between the section elements on any processor is $\text{lcm}(s_2, P_2) = 6$.

For each processor $q$ in $PSend(p)$, there exists a corresponding first slice index $k_1$ in $(i_1 : \min(j_1, i_1 + \frac{\text{lcm}(s_2, P_2)}{s_2} - 1))$. The indices of all slices sent by $p$ to $q$ can be determined as $(k_1 : j_1 : \frac{\text{lcm}(s_2, P_2)}{s_2})$. Since $DSend(p, q)$ consists of the local indices of these slices on processor $p$, we have

$$DSend(p, q) = l_1 + \left( k_1 : j_1 : \frac{\text{lcm}(s_2, P_2)}{s_2} \right) \cdot s_1 - m_1 - pb_1$$

$$= \left( l_1 + k_1 \cdot s_1 - m_1 - p \cdot b_1 : l_1 + j_1 \cdot s_1 - m_1 - p \cdot b_1 : \frac{\text{lcm}(s_2, P_2) \cdot s_1}{s_2} \right). \hspace{1cm} (3.25)$$

We now consider the receive processor set of $p$. The elements of array $B$ located on processor $p$ are $B(m_2 + p : n_2 : P_2)$. Processor $p$ may contain elements of the array
Figure 17: Closed forms for block to cyclic case ($P_2 = 3, s_2 = 2$).

section $B(l_2 : u_2 : s_2)$ only if the Diophantine equation $l_2 + is_2 = m_2 + p + cP_2$ has a solution, i.e., $\gcd(s_2, P_2)$ divides $(p + m_2 - l_2)$ [23, 139]. In this case, $p$ may receive array elements of $A$ from other processors. To determine the first array slice located on $p$, we solve $i_2s_2 - c_2P_2 = m_2 + p - l_2$ using Euclid’s Extended GCD algorithm [23, 81]. Let $i_2$ and $c_2$ be the solution such that $i_2$ is the smallest non-negative integer for which the corresponding $c_2$ is also a non-negative integer. The first slice index is $i_2$ and the last slice index $j_2$ is computed as follows:

$$j_2 = i_2 + \left\lfloor \frac{u_2 - l_2 - i_2s_2}{\operatorname{lcm}(s_2, P_2)} \right\rfloor \times \frac{\operatorname{lcm}(s_2, P_2)}{s_2}. \quad (3.26)$$

The indices of slices of $B(l_2 : u_2 : s_2)$ located on processor $p$ are $(i_2 : j_2 : \frac{\operatorname{lcm}(s_2, P_2)}{s_2})$.

To determine the receive processor set, we need to compare the block size $b_1$ with the
stride distance between two consecutive elements of $A(l_1 : u_1 : s_1)$ that are sent to processor $p$. The stride distance is $\frac{s_1 \cdot \text{lcm}(s_2, P_2)}{s_2}$, as shown in Fig. 18. If the stride distance is less than $b_1$, then $p$ will receive data elements from processors $((i_2 s_1 + l_1 - m_1) \div b_1 : (j_2 s_1 + l_1 - m_1) \div b_1)$. If the stride is not less than $b_1$, then $p$ will receive one data element each from the processors in $\left( (i_2 : j_2 : \frac{\text{lcm}(s_2, P_2)}{s_2}) \cdot s_1 + l_1 - m_1 \right) \div b_1$. Thus, we have:

$$P\text{Recv}(p) = \begin{cases} 
\left( (l_1 + i_2 s_1 - m_1) \div b_1 : (j_2 s_1 + l_1 - m_1) \div b_1 \right) 
& \text{if } i_2 \leq j_2 \land \frac{s_1 \cdot \text{lcm}(s_2, P_2)}{s_2} < b_1, \\
\left( l_1 + (i_2 : j_2 : \frac{\text{lcm}(s_2, P_2)}{s_2}) \cdot s_1 - m_1 \right) \div b_1 
& \text{if } i_2 \leq j_2 \land \frac{s_1 \cdot \text{lcm}(s_2, P_2)}{s_2} \geq b_1, \\
\emptyset 
& \text{if } i_2 > j_2.
\end{cases} \tag{3.27}$$

To determine the receive data index set of processor $p$ from $q$, we need the first
and last slice indices \(i_r\) and \(j_r\), respectively, of the data index set. The evaluation of \(i_r\) is shown in Fig. 19. These indices are computed as below:

\[
i_r = i_2 + \left[ \frac{\max(m_1 + qb_1 - l_1 - i_2 s_1, 0) s_2}{s_1 \times \text{lcm}(s_2, P_2)} \right] \times \frac{\text{lcm}(s_2, P_2)}{s_2}, \tag{3.28}
\]

\[
j_r = i_2 + \left[ \frac{(m_1 + qb_1 + b_1 - 1 - l_1 - i_2 s_1) s_2}{s_1 \times \text{lcm}(s_2, P_2)} \right] \times \frac{\text{lcm}(s_2, P_2)}{s_2}. \tag{3.29}
\]

Thus, the receive data index set of processor \(p\) from \(q\) is:

\[
D\text{Recv}(p, q) = \left( l_2 + i_r s_2 - m_2 \right) \text{ div } P_2 : \left( l_2 + j_r s_2 - m_2 \right) \text{ div } P_2 : \frac{\text{lcm}(s_2, P_2)}{P_2}. \tag{3.30}
\]

Node code for processor \(p\), can be similarly developed using these closed forms.
3.2.3 Cyclic Distribution to Block Distribution

The case when $A(m_1 : n_1)$ is cyclically distributed and $B(m_2 : n_2)$ is block distributed is the dual of the previous case where $A$ was block distributed and $B$ cyclically distributed. The send processor and send data index sets of the former become the receive processor and receive data index sets of the latter. Similarly, the receive sets of the former become the send sets of the latter.

3.2.4 Cyclic Distribution to Cyclic Distribution

We now consider the case where arrays $A$ and $B$ are cyclically distributed over $P_1$ and $P_2$ processors, respectively. The section indices $i_1$ and $j_1$ of the first and last element of $A(u_1 : s_1)$ located on processor $p$ are determined as follows. Let $i_1$ and $c_1$ be the solution to the Diophantine equation $i_1 s_1 - c_1 P_1 = m_1 + p - l_1$ such that $i_1$ is the smallest non-negative integer for which the corresponding $c_1$ is also a non-negative integer. Let $r_1 = \text{lcm}(s_1, P_1)$. Then index $j_1$ is:

$$j_1 = i_1 + \left\lfloor \frac{u_1 - l_1 - i_1 s_1}{r_1} \right\rfloor \times \frac{r_1}{s_1}. \quad (3.32)$$

Let $t_2 = \text{lcm}\left(\frac{r_1}{s_1}, P_2\right)$. The send processor set is defined as below:

$$P_{\text{Send}}(p) = \left( l_2 + \left( i_1 : \min \left( j_1, i_1 + \frac{t_2}{s_2} - 1 \right) : \frac{r_1}{s_1} \right) \times s_2 - m_2 \right) \mod P_2 \quad (3.33)$$

For each processor $q$ in $P_{\text{Send}}(p)$, there exists a corresponding first slice index $k_1$ in $\left( i_1 : \min \left( j_1, i_1 + \frac{t_2}{s_2} - 1 \right) : \frac{r_1}{s_1} \right)$. The indices of all slices sent to $q$ can be determined as $\left( k_1 : j_1 : \frac{t_2}{s_2} \right)$. However, the data index send set of processor $p$ to processor $q$ is
defined as the set of local indices on $p$ of those slices of array $A$:

$$DSend(p, q) =$$

$$\left( (l_1 + k_1 \ast s_1 - m_1) \div P_1 : (l_1 + j_1 \ast s_1 - m_1) \div P_1 : \frac{t_2 \ast s_1}{s_2 \ast P_1} \right)$$

Determining the communication sets for the receiving phase for cyclic distribution to cyclic distribution is a dual of the problem of the sending phase. Let $i_2$ and $c_2$ be the solution to the Diophantine equation $i_2 s_2 - c_2 P_2 = m_2 + p - l_2$ such that $i_2$ is the smallest non-negative integer for which the corresponding $c_2$ is also non-negative.

Let $r_2 = \text{lcm}(s_2, P_2)$ and $t_1 = \text{lcm}\left(\frac{ra-s_1}{s_2}, P_1\right)$. Then we have:

$$j_2 = i_2 + \left[ \frac{u_2 - l_2 - i_2 s_2}{r_2} \right] \ast \frac{r_2}{s_2},$$

$$k_2 \in \left( i_2 : \min \left( j_2, i_2 + \frac{t_1}{s_1} - 1 \right) : \frac{r_2}{s_2} \right),$$

$$PRecv(p) = \left( l_1 + \left( i_2 : \min \left( j_2, i_2 + \frac{t_1}{s_1} - 1 \right) : \frac{r_2}{s_2} \right) \ast s_1 - m_1 \right) \mod P_1 \quad (3.38)$$

The data receive index set is given by

$$DRecv(p, q) =$$

$$\left( (l_2 + k_2 \ast s_2 - m_2) \div P_2 : (l_2 + j_2 \ast s_2 - m_2) \div P_2 : \frac{t_1 \ast s_2}{s_1 \ast P_2} \right)$$

Node code for processor $p$, can be similarly developed using these closed forms.

### 3.2.5 Modifications for Negative Strides

We now describe the modifications to the closed form expressions to handle negative strides. If both $s_1 < 0$ and $s_2 < 0$ then the array statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ is equivalent to the array statement $B(u_2' : l_2 : -s_2) = f(A(u_1' : l_1 : -s_1))$, ...
where \( u'_2 = u_2 + (l_2 - u_2) \mod s_2 \) and \( u'_1 = u_1 + (l_1 - u_1) \mod s_1 \). Since \(-s_1 > 0\) and \(-s_2 > 0\), the communication sets for the array statement \( B(u'_2 : l_2 : -s_2) = f(A(u'_1 : l_1 : -s_1)) \) can be evaluated using the techniques presented in the previous sections.

The cases when \( s_1 \) and \( s_2 \) differ in sign are slightly more complicated as the direction in which the slice indices of the source and the target sections increase are different. Consider the case when \( s_1 < 0 \) and \( s_2 > 0 \). Closed form expressions for the communication sets for these cases are developed by inverting the numbering of the slice indices of section \( A(l_1 : u_1 : s_1) \), noting that the slice index \( i_1 \) for the array section \( A(l_1 : u_1 : s_1) \) corresponds to the slice index \( k - i_1 \) in the array section \( B(l_2 : u_2 : s_2) \), where \( k = \left\lfloor \frac{u_2 - l_1}{s_2} \right\rfloor \) is the total number of slice indices in the array sections, and then following arguments similar to those in the previous sections. For instance, consider the case when both \( A \) and \( B \) are distributed using a block distribution. Let processor \( p \) be the sending processor and \( A(l_1 + i_1 s_1) \) and \( A(l_1 + j_1 s_1) \) be the first and last elements of \( A(l_1 : u_1 : s_1) \) allocated to \( p \). We obtain \( i_1 \) and \( j_1 \) as follows:

\[
\begin{align*}
i_1 &= \left\lfloor \frac{\max(m_1 + pb_1 - u'_1, 0)}{-s_1} \right\rfloor, \quad (3.40) \\
j_1 &= \left\lfloor \frac{\min(m_1 + pb_1 + b_1 - 1 - u'_1, l_1 - u_1)}{-s_1} \right\rfloor, \quad (3.41)
\end{align*}
\]

where \( u'_1 \) is as defined above. The send processor set of \( p \) includes processors that own elements in \( B(l_2 + (k - j_1) s_2 : l_2 + (k - i_1) s_2 : s_2) \). Note that the slice indices in the array section \( B(l_2 : u_2 : s_2) \) corresponding to \( i_1 \) and \( j_1 \) are \( k - i_1 \) and \( k - j_1 \), respectively. Thus depending on the relation between the stride \( s_2 \) and the block size
Table 2: Virtualization schemes.

<table>
<thead>
<tr>
<th>Source Distribution Virtual View</th>
<th>Target Distribution Virtual View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual-Block</td>
<td>Block to Block</td>
</tr>
<tr>
<td>Virtual-Cyclic</td>
<td>Cyclic to Block</td>
</tr>
</tbody>
</table>

\[
P_{Send}(p) = \begin{cases} 
((l_2 + (k - j_1)s_2 - m_2) \div b_2 : (l_2 + (k - i_1)s_2 - m_2) \div b_2) & \text{if } i_1 \leq j_1 \land s_2 < b_2, \\
((k - j_1) : (k - i_1)) * s_2 + l_2 - m_2) \div b_2 & \text{if } i_1 \leq j_1 \land s_2 \geq b_2, \\
\emptyset & \text{if } i_1 > j_1.
\end{cases}
\]

The data send set \(D_{Send}(p, q)\) for \(q \in P_{Send}(p)\) can be similarly evaluated.

3.3 Virtual Processor Approach for Block-Cyclic Distributions

In this section, we present a virtual processor approach for efficient execution of array statements involving block-cyclically distributed arrays. Let \(A(m_1 : n_1)\) and \(B(m_2 : n_2)\) be distributed using a cyclic\((b_1)\) and cyclic\((b_2)\) distribution on \(P_1\) and \(P_2\) processors, respectively. For an array statement of the form \(B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))\), the virtual processor approach involves:

1. Viewing a cyclic\((b_1)\) distribution of \(A\) as a block (or cyclic) distribution on \(VP_1\) virtual processors which are cyclically (or block-wise) mapped to \(P_1\) processors.

These views are referred to as virtual-block or virtual-cyclic views depending on whether a block or cyclic distribution of the array on the virtual processors is
used. Fig. 20 gives an schematic illustration of the two views of a block-cyclic distribution. The cyclic\(^{(b)}\) distribution of \(B\) is similarly viewed as a block or cyclic distribution on \(VP_2\) virtual processors.

2. The communication required to perform the array statement can be determined by using the closed forms of Section 3.2 in the virtual processor domain. Each physical processor bears the responsibility of performing the computation and communication for the virtual processors mapped to it.

Thus the virtual processor approach is characterized by a two-level mapping of array elements to physical processors. The first level maps the array elements to virtual processors and the second level maps virtual processors to physical processors.
The mapping at each level can be represented using simple regular sections which facilitates efficient implementation of this approach.

We now present the details of the virtualization schemes. Depending upon the virtualization views used for the source array $A$ and the target array $B$, four different schemes for executing the array statement are possible. The four schemes are shown in Table 2. Each scheme is associated with a different number of source and target virtual processors, a different communication pattern in the virtual processor domain, and incurs different indexing overheads. Hence, we also present a strategy to select the communication scheme with minimum indexing overhead.

### 3.3.1 Virtualization Views

We now describe the virtual cyclic and virtual block views of a block-cyclic distribution.

**Virtual-Block View**

Let array $A(m : n)$ be distributed using a $cyclic(b)$ distribution on $P$ processors. In the *virtual-block view*, $A$ is assumed to be block distributed on $VP = \lceil \frac{n-m+1}{b} \rceil$ virtual processors. These virtual processors are assigned to $P$ processors in a cyclic fashion. The set of virtual processors on processor $p$ is $(p : VP : P)$. Fig. 21 illustrates the virtual-block view of a $cyclic(2)$ distribution of $A(0 : 15)$ on two processors. The array has a block distribution on eight virtual processors $v_0$ to $v_7$, which are cyclically allocated to the two processors $p_0$ and $p_1$, i.e., $v_0$, $v_2$, $v_4$ and $v_6$ are mapped to $p_0$, while $v_1$, $v_3$, $v_5$ and $v_7$ are mapped to $p_1$. Processor $p_0$ will perform the computation...
and communication for virtual processors $v_0$, $v_2$, $v_4$ and $v_6$ while processor $p_1$ will perform the same for virtual processors $v_1$, $v_3$, $v_5$ and $v_7$.

Using closed form expressions developed in Section 3.2, data index sets can be evaluated in terms of the local indices of the virtual processors. However, the local index of an element on a virtual processor is not the same as its local index on the processor to which it is mapped. For instance, in Fig. 21, element $A(8)$ has a local index of 4 in $A_{loc}$ on processor $p_0$, but a local index of 0 on virtual processor $v_4$. Since the physical processor $p$ performs the computation and communication for the virtual processor $v$ mapped to it, it is necessary to determine the translation from the virtual processor's local index space to the physical processor's index space. If the virtual processor $v$ is mapped to processor $p$, then the array element with local index $j$ on $v$ has a local index $(v \div P) \cdot b + j$ on processor $p$. Under the virtual-block view, a stride of $s$ in the local index space of a virtual processor on processor $p$ remains unchanged in the local index space of $p$. Hence, an array section $[l : u : s)$ of $A$ in the local space of a virtual processor $v$ on processor $p$ corresponds to array section $[(v \div P) \cdot b + l : (v \div P) \cdot b + u : s)$.

Consider the array section $A(l : u : s)$. Under the virtual-block view not all virtual processors necessarily own elements of the array section. We refer to virtual processors that own array section elements as being active. The active virtual processors are determined as follows. If $s \leq b$, then each virtual processor $v \in ((l - m) \div b : (u - m) \div b)$ has at least one element of the array section. Let $v_l = (l - m) \div b$ and $v_u = (u - m) \div b$. Since the set of virtual processors on processor $p$ is $(p : VP : P)$,
Figure 21: Virtual-block view of array $A(0 : 15)$ with cyclic(2) distribution on two processors.

The active virtual processors on $p$ for array $A$, denoted by $VAct.A(p)$, are given by

$$ VAct.A(p) = (v_l : v_u) \cap (p : VP : P) $$

$$ = (\max(v_l + (p - v_l) \mod P, p) : \min(v_u, VP) : P) $$

If $s > b$, then each active virtual processor has exactly one element of the array section. The active virtual processors are given by $(((l - m) : (u - m) : s) \div b) \cap (p : VP : P)$. Each active virtual processor in $(((l - m) : (u - m) : s) \div b$ is scanned and a check performed to determine if it is located on processor $p$.

**Virtual-Cyclic View**

Under the virtual-cyclic view, array $A(m : n)$ distributed using a cyclic($b$) distribution is assumed to have a cyclic distribution on $VP = \min(P \ast b, n - m + 1)$ virtual processors. These virtual processors are block distributed on the $P$ processors. Thus, the set of virtual processors on a processor $p$ are $(p \ast b : \min(p \ast b + b - 1, n - m + 1))$. Fig. 22 illustrates the virtual-cyclic view of a cyclic(2) distribution of $A(0 : 15)$ on two processors. The array has a cyclic distribution on four virtual processors. These
Figure 22: Virtual-cyclic view of array $A(0 : 15)$ with $cyclic(2)$ distribution on two processors.

Four virtual processors are allocated to the two processors in a block fashion, i.e., $v_0$ and $v_1$ are mapped to $p_0$, while $v_2$ and $v_3$ are mapped to $p_1$.

The array element with local index $j$ on a virtual processor $v$, has a local index $(v \mod b) + b \cdot j$ on the processor to which it is mapped. A stride of $s$ in the local index space of a virtual processor corresponds to $s \cdot b$ in the processor's local index space. Thus the array section $(l : u : s)$ in the local index space of a virtual processor $v$ is mapped to the section $((v \mod b) + b \cdot l : (v \mod b) + b \cdot u : s \cdot b)$ in the local index space of the processor it is mapped to.

Consider the array section $A(l : u : s)$. The active virtual processors on a processor $p$ for the array section are determined as follows. The elements of $A$ located on a virtual processor $v$ have indices $(v + m : n : P \cdot b)$. Thus $v$ is active if the intersection
(v + m : n : P * b) \cap (l : u : s) is not empty. Hence, a virtual processor v is active if \( \gcd(s, P * b)(v + m - l) \) and the first element of the intersection lies in the array section. The first element can be found by solving the Diophantine equation \( i * s - c * P * b = v + m - l \). Let \( i_1 \) and \( c_1 \) be the solution such that \( i_1 \) is the smallest non-negative integer for which the corresponding \( c_1 \) is non-negative. If \( l + i_1 * s \leq u \) then the virtual processor v is active. If the first active virtual processor on processor \( p \) is \( v_f \), then the active virtual processors on \( p \) are

\[
V\text{Act-A}(p) \subseteq (v_f : \min(u - m, p * b + b - 1) : \gcd(s, P * b)) \tag{3.45}
\]

Note that not every processor in \((v_f : \min(u - m, p * b + b - 1) : \gcd(s, P * b))\) will have an element of the array section. The first virtual processor \( v_f \) is given by \( v_f = ((-p * b - m + l) \mod \gcd(s, P * b)) + p * b \).

### 3.3.2 Virtualization Schemes

We now present virtualization schemes which use virtualized views for the source and target arrays and the communication sets developed in Section 3.2 for block and cyclic distributions, to determine the communication sets for \( cyclic(b) \) distributions.

The send and receive processor sets and data index sets for a virtual processor \( v_p \), which is mapped to processor \( p \) are defined as follows:

\[
V\text{PSend}(v_p) = \text{Set of virtual processors } \{v\}, \text{ to which } v_p \text{ sends data.} \tag{3.46}
\]

\[
V\text{PRecv}(v_p) = \text{Set of virtual processors } \{v\}, \text{ from which } v_p \text{ receives data.} \tag{3.47}
\]
\[ VD\text{Send}(v_p, v_q) = \text{Local indices on processor } p \text{ of the data elements,} \]
\[ \quad \text{which } v_p \text{ sends to } v_q. \]  
(3.48)

\[ VD\text{Recv}(v_p, v_q) = \text{Local indices on processor } p \text{ of the data elements} \]
\[ \quad \text{which } v_p \text{ receives from } v_q. \]  
(3.49)

\( VPSend(v_p) \) and \( VP\text{Recv}(v_p) \) are obtained from \( P\text{Send()} \) and \( P\text{Recv()} \) assuming a block or cyclic distribution on the virtual processors. Similarly, \( VD\text{Send}(v_p, v_q) \) and \( VD\text{Recv}(v_p, v_q) \) are obtained from \( D\text{Send()} \) and \( D\text{Recv()} \). Note that \( VD\text{Send()} \) and \( VD\text{Recv()} \) are defined in terms of the local index space of the physical processor and a translation from the virtual processor local index space to physical index space will be required. Node pseudo code for execution of the array statement \( B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1)) \) using the processor sets and data index sets in the virtual processor domain is shown in Fig. 23. \( Phy.A(v) \) denotes the processor to which virtual processor \( v \) is mapped under the virtual view for the array \( A(m_1 : n_1) \). Similarly, \( Phy.B(v) \) denotes the processor to which virtual processor \( v \) is mapped under the virtual view for the array \( B(m_2 : n_2) \).

The node pseudo code in Fig. 23 could be inefficient as it may involve sending multiple messages between two processors. The additional message startup overhead can be reduced by splitting the send and receive phase into two phases. First, \( P\text{Send}(p) \) and \( D\text{Send}(p, q) \) are evaluated by scanning the active virtual processors and determining the processor and data send and receive index sets. These sets are defined as
follows:

\[
PSend(p) = \bigcup_{vp \in VAct-A(p)} \bigcup_{vq \in VPSend(vp)} Phy.A(v_q) \tag{3.50}
\]

\[
PRect(p) = \bigcup_{vp \in VAct-B(p)} \bigcup_{vq \in VPRect(vp)} Phy.A(v_q) \tag{3.51}
\]

\[
DSend(p, q) = \bigcup_{vp \in VAct-A(p)} \bigcup_{vq \in (VAct-B(q) \cap VPSend(vp))} VDSend(v_p, v_q) \tag{3.52}
\]

\[
DRect(p, q) = \bigcup_{vp \in VAct-B(p)} \bigcup_{vq \in (VAct-A(q) \cap VPRect(vp))} VDRect(v_p, v_q) \tag{3.53}
\]

where \( VAct.A(p) \) denotes the set of active virtual processors on processor \( p \) corresponding to the section \( A(l_1 : u_1 : s_1) \) and \( VAct.B(p) \) denotes the set of active virtual processors on processor \( p \) corresponding to the section \( B(l_2 : u_2 : s_2) \). The node pseudo code using these sets is as shown in Fig. 24 and Fig. 25. The initialization of the appropriate sets is not shown in the figure. The first phase evaluates the processor and data index send and receive sets. Note that the Phase two packs and sends the data to each processor in \( PSend(p) \), receives messages from all processors \( q \in PRect(p) \) and evaluates the new values for \( B \) using \( DRect(p, q) \).

The message contains all data communicated from processor \( p \) to \( q \), i.e., data from all active virtual processors on \( p \) to their target virtual processors on \( q \). To facilitate unpacking, the data is packed in increasing order of target virtual processor index. Data for a particular target virtual processor from all its source virtual processors on processor \( p \) is stored in the increasing order of source virtual processor index.

### 3.3.3 Strategy for Selection of Virtualization Schemes

Given block-cyclic distributions for \( A(m_1 : n_1) \) and \( B(m_2 : n_2) \), four choices are available for the virtualization scheme used, as shown in Table 2. The choice of
/* Node Code for Processor $p$ */

/* Sending Phase */

for $v_p \in V Act_A(p)$
  for $v_q \in V PSend(v_p)$
    $tmp.out(0 : |V DSend(v_p, v_q)| - 1) = A.loc(V DSend(v_p, v_q))$
    send($Phy.B(v_q), tmp.out$)
  endfor
endfor

/* Receiving Phase */

for $v_p \in V Act_B(p)$
  for $v_q \in V PRecv(v_p)$
    recv($Phy_A(v_q), tmp.in$)
    $B.loc(V DRecv(v_p, v_q)) = f(tmp.in)$
  endfor
endfor

Figure 23: Communication phases in the virtual processor domain.
/* Node Code for Processor p */

/* Sending Phase 1 */
for vp ∈ VActA(p)
    for vq ∈ VPSend(vp)
        q = PhyB(vq)
        PSend(p) = PSend(p) ∪ {q}
        DSend(p, q) = DSend(p, q) ∪ VDSend(vp, vq)
    endfor
endfor

/* Sending Phase 2 */
for q ∈ PSend(p)
    tmp_out(0 : |DSend(p, q)| − 1) = Aloc(DSend(p, q))
    send(q, tmp_out)
endfor

Figure 24: Template for node code with virtual processor approach: Send Phase.
/* Node Code for Processor p */

/* Receiving Phase 1 */
for vp ∈ VAct.B(p)
  for vq ∈ VRecv(vp)
    q = Phy.A(vq)
    PRecv(p) = PRecv(p) ∪ {q}
    DRecv(p, q) = DRecv(p, q) ∪ VDRecv(vp, vq)
  endfor
endfor

/* Receiving Phase 2 */
for q ∈ PRecv(p)
  recv(q, tmp.in)
  B.loc(DRecv(p, q)) = f(tmp.in)
endfor

Figure 25: Template for node code with virtual processor approach: Receive phase.
the virtualization scheme depends on the additional indexing overhead per processor. Since the communication pattern between physical processors is identical for all four schemes, the scheme with lowest indexing overhead would be expected to have the lowest total completion time. During the send phase, a processor \( p \) packs all the data in \( D_{Send}(p, q), \forall q \in P_{Send}(p) \). As shown in Section 3.3.2, \( D_{Send}(p, q) \) is a union of array sections. Associated with each array section is the time for evaluation of the loop bounds and the loop overhead. Assuming that this overhead is nearly equal for each section, the total indexing cost per processor during the sending phase, \( t_s(p) \), is proportional to the total number of array sections to be communicated.

\[
t_s(p) = t_i \left( \sum_{v \in V_{Act-A}(p)} |V_{P_{Send}}(v)| \right) \leq t_i \cdot V_{Act-A_{max}} \cdot V_{P_{Send}_{max}} \quad (3.54)
\]

where \( V_{Act-A_{max}} \) and \( V_{P_{Send}_{max}} \) are defined as

\[
V_{Act-A_{max}} = \max_{0 \leq p < P_1} (|V_{Act-A}(p)|), \quad (3.55)
\]

\[
V_{P_{Send}_{max}} = \max_{0 \leq v < V_{P_1}} (|V_{P_{Send}}(v)|), \quad (3.56)
\]

and \( V_{P_1} \) is the total number of virtual processors corresponding to the virtual view of the source distribution, and \( t_i \) is a measure of the indexing overhead per section.

Similarly, for the total indexing cost per processor during the receiving phase, \( t_r(p) \), we have

\[
t_r(p) = t_i \left( \sum_{v \in V_{Act-B}(p)} |V_{P_{Recv}}(v)| \right) \leq t_i \cdot V_{Act-B_{max}} \cdot V_{P_{Recv}_{max}} \quad (3.57)
\]

where \( V_{Act-B_{max}} \) and \( V_{P_{Recv}_{max}} \) are defined as

\[
V_{Act-B_{max}} = \max_{0 \leq q < P_2} (|V_{Act-B}(q)|), \quad (3.58)
\]

\[
V_{P_{Recv}_{max}} = \max_{0 \leq v_q < V_{P_2}} (|V_{P_{Recv}}(v_q)|), \quad (3.59)
\]
Table 3: Send and receive processor set size estimates for various source and target distributions.

<table>
<thead>
<tr>
<th></th>
<th>$VPSend_{\max}$</th>
<th>$VPRecv_{\max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block to Block</strong></td>
<td>$\frac{b_1+s_2}{b_2+s_1}$, if $s_2 &lt; b_2$</td>
<td>$\frac{b_2+s_1}{b_1+s_2}$, if $s_1 &lt; b_1$</td>
</tr>
<tr>
<td></td>
<td>$\frac{b_1}{s_1}$, if $s_2 \geq b_2$</td>
<td>$\frac{b_2}{s_2}$, if $s_1 \geq b_1$</td>
</tr>
<tr>
<td><strong>Block to Cyclic</strong></td>
<td>min($\frac{b_1}{s_1}, \frac{r_2}{s_2}$)</td>
<td>$\frac{c_2<em>P_2+s_1}{b_1+s_2}$, if $\frac{s_1</em>P_2}{s_2} &lt; b_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\frac{c_2<em>P_2}{r_2}$, if $\frac{s_1</em>P_2}{s_2} \geq b_1$</td>
</tr>
<tr>
<td><strong>Cyclic to Block</strong></td>
<td>$\frac{c_1<em>P_1+s_2}{b_2+s_1}$, if $\frac{s_2</em>P_1}{s_1} &lt; b_2$</td>
<td>min($\frac{b_2}{s_2}, \frac{r_1}{s_1}$)</td>
</tr>
<tr>
<td></td>
<td>$\frac{c_1<em>P_1}{r_1}$, if $\frac{s_2</em>P_1}{s_1} \geq b_2$</td>
<td></td>
</tr>
<tr>
<td><strong>Cyclic to Cyclic</strong></td>
<td>min($\frac{c_1<em>P_1}{r_1}, \frac{lcm(r_1</em>\frac{s_2*P_1}{s_1})}{r_1}$)</td>
<td>min($\frac{c_2<em>P_2}{r_2}, \frac{lcm(r_2</em>\frac{s_1*P_1}{s_2})}{r_2}$)</td>
</tr>
</tbody>
</table>

$c_1 = \frac{N_1}{P_1b_1}$, $c_2 = \frac{N_2}{P_2b_2}$, $r_1 = lcm(s_1, P_1)$, $r_2 = lcm(s_2, P_2)$

and $VP_2$ is the total number of virtual processors corresponding to the target distribution. Since $t_i$ is a constant, $VAct.A_{\max} \cdot VPSend_{\max} + VAct.B_{\max} \cdot VPRecv_{\max}$ is used as the cost estimate of the indexing overhead.

Using the closed forms in Section 3.2, approximate measures for $VPSend_{\max}$ and $VPRecv_{\max}$ can be obtained. These measures are shown in Table 3. The source array of size $N_1$ is distributed on $P_1$ processors using a cyclic($b_1$) distribution and the target array of size $N_2$ is distributed on $P_2$ processors using a cyclic($b_2$) distribution. Approximate measures for the maximum number of virtual processors per physical
processor are

$$V_{Act_{\text{max}}} = \frac{N}{P \cdot \max(s, b)}$$

(3.60)

for the virtual-block view and

$$V_{Act_{\text{max}}} = \frac{b}{\gcd(s, Pb)}$$

(3.61)

for the virtual-cyclic view. Thus given a source and target block-cyclic distribution, an approximation of the maximum indexing cost can be obtained for each of the four virtualization schemes and a choice of the scheme best suited for the given distributions can be made.

### 3.4 Index Sets for Multi-Dimensional Arrays

We have presented techniques for handling one-dimensional arrays. The virtual processor approach can be extended to handle multi-dimensional arrays. The communication sets are developed independently for each array dimension using the techniques presented for one-dimensional arrays. The communication sets for the array statement involving multi-dimensional arrays are obtained by taking the cross-product of the appropriate sets for each dimension. Consider arrays $A(m_{a1} : n_{a1}, m_{a2} : n_{a2})$ and $B(m_{b1} : n_{b1}, m_{b2} : n_{b2})$ distributed using block-cyclic distributions on a $P_1 \times P_2$ processor mesh. The processors in the mesh are labeled as $(p_1, p_2)$, $0 \leq p_1 < P_1$ and $0 \leq p_2 < P_2$. Consider the array statement $B(l_{b1} : u_{b1}, l_{b2} : u_{b2} : s_{b2}) = f(A(l_{a1} : u_{a1}, l_{a2} : u_{a2} : s_{a2}))$. The active virtual processors on processor $(p_1, p_2)$ are $V P A c t_{\cdot A1}(p_1) \otimes V P A c t_{\cdot A2}(p_2)$ where $V P A c t_{\cdot A1}(p_1)$ and $V P A c t_{\cdot A2}(p_2)$ are the
sets of active virtual processors corresponding to the first and second dimensions of array $A$, respectively and $\otimes$ denotes the cross-product of two sets. The sets $VP_{Send}$, $VP_{Recv}$, $VD_{Send}$, and $VD_{Recv}$ for the multi-dimensional array are defined similarly. Thus the communication code for the multi-dimensional array statement would be as shown in Fig. 26 and Fig. 27. The send and receive sets for the first dimension have a subscript one, while those for the second dimension have a subscript two. This communication code should be expressed as two-phases to eliminate multiple messages between two physical processors. Note that the closed form expressions developed for the communication sets in Section 3.2 allow for a different number of physical processors at the source and target distribution. Thus a change of the shape of the underlying processor grid and the multi-dimensional array can be handled by the virtual processor approach.

3.5 Performance Results

In this section, we present experimental results for the virtual processor approach. The experiments were performed on a 32-node Cray T3D. The node programs used the PVM message-passing library. The time required by each processor to execute the node program for the array assignment statement was measured and the maximum time among all processors reported. Times were measured using the rtclock() wall clock timer.

The goals of the performance measurement were to determine whether the indexing costs for the four virtualization schemes were significantly different and whether
/* Node Code for Processor \((p_1, p_2)\) */

/* Sending Phase */

\[
\begin{align*}
&\text{for } v_{p1} \in VAct.A1(p_1) \\
&\quad \text{for } v_{p2} \in VAct.A2(p_2) \\
&\quad\quad \text{for } v_{q1} \in VPSend_1(v_{p1}) \\
&\quad\quad\quad \text{for } v_{q2} \in VPSend_2(v_{p2}) \\
&\quad\quad\quad\quad \text{packA\_loc(VDSend}_1(v_{p1}, v_{q1}), VDSend}_2(v_{p2}, v_{q2}) \text{ in tmp\_out} \\
&\quad\quad\quad\quad \text{send}((\text{Phy\_B1}(v_{q1}), \text{Phy\_B2}(v_{q2})), \text{tmp\_out})
\end{align*}
\]

endfor endfor endfor endfor

Figure 26: Send phase for two-dimensional arrays in the virtual processor domain.

/* Node Code for Processor \((p_1, p_2)\) */

/* Receiving Phase */

\[
\begin{align*}
&\text{for } v_{p1} \in VAct.B1(p_1) \\
&\quad \text{for } v_{p2} \in VAct.B2(p_2) \\
&\quad\quad \text{for } v_{q1} \in VPRcv_1(v_{p1}) \\
&\quad\quad\quad \text{for } v_{q2} \in VPRcv_2(v_{p2}) \\
&\quad\quad\quad\quad \text{recv}((\text{Phy\_A1}(v_{q1}), \text{Phy\_A2}(v_{q2})), \text{tmp\_in}) \\
&\quad\quad\quad\quad \text{B\_loc(VDRecv}_1(v_{p1}, v_{q1}), VDRecv}_2(v_{p2}, v_{q2})) = f(\text{tmp\_in})
\end{align*}
\]

endfor endfor endfor endfor

Figure 27: Receive phase for two-dimensional arrays in the virtual processor domain.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>vb-vb</th>
<th>vb-vc</th>
<th>vc-vb</th>
<th>vc-vc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>IE</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3000</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14400</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>2240</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>930</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>1</td>
<td>1</td>
<td>449*</td>
</tr>
<tr>
<td>5</td>
<td>7200</td>
<td>1</td>
<td>1</td>
<td>2880</td>
</tr>
<tr>
<td>225</td>
<td>7200</td>
<td>1</td>
<td>1</td>
<td>64*</td>
</tr>
<tr>
<td>7200</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4800</td>
</tr>
</tbody>
</table>

*: predicted best., †: actual best.

the developed heuristic provided a good indication of the scheme with lowest indexing cost. Further, an estimate of the table generation costs and a comparison of the table generation costs with the execution time for the array statement was desired. Also a verification of the premise that the scheme with the lowest indexing cost also has the lowest execution time and table generation time was desired.

To reduce the number of independent parameters, we measured times for array section assignments of the form $B(0 : s_1 * M - 1 : s_1) = A(0 : s_2 * M - 1 : s_2)$. Double precision arrays $A$ and $B$ are distributed using cyclic($b_1$) and cyclic($b_2$) distributions, respectively. The parameter $M$ gives the number of active elements in each array section. For our experiments we kept the number of active elements $M$ fixed at 230400 and $P$ at 32 processors. The experiment was performed for three data sets categorized by the relative values of $s_1$ and $s_2$:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>vb-vb</th>
<th>vb-vc</th>
<th>vc-vb</th>
<th>vc-vc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IE IT</td>
<td>IE IT</td>
<td>IE IT</td>
<td>IE IT</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>9000</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>14400</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>6720</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>3720</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>2</td>
<td>2</td>
<td>898*</td>
</tr>
<tr>
<td>5</td>
<td>21600</td>
<td>3</td>
<td>3</td>
<td>8640</td>
</tr>
<tr>
<td>225</td>
<td>36000</td>
<td>5</td>
<td>5</td>
<td>320*</td>
</tr>
<tr>
<td>14400</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>9600</td>
</tr>
</tbody>
</table>

*: predicted best., †: actual best.

(a) Both the source and target array sections have unit strides, i.e. $s_1 = s_2 = 1$.

This case corresponds to array redistribution.

(b) Both the source and target array sections have the same non-unit stride, $s_1 = s_2 > 1$.

(c) The source and target array sections have different strides, i.e. $s_1 \neq s_2$.

Table 4, Table 5, and Table 6 present the indexing cost estimates (IE) for the four virtualization schemes and the corresponding indexing times (IT). The indexing time includes the time for copying the data into and out of message buffers and does not include the time spent for communication. In these tables, vb-vb refers to the virtual block to virtual block scheme, vb-vc refers to the virtual block to virtual cyclic scheme, vc-vb refers to the virtual cyclic to virtual block scheme and vc-vc refers to
Table 6: Evaluation of the strategy for the virtualization scheme selection \((P = 32, M = 230400)\). IE: Indexing Cost Estimate, IT: Indexing Time (ms).

<table>
<thead>
<tr>
<th>Parameters</th>
<th>vb-vb</th>
<th>vb-vc</th>
<th>vc-vb</th>
<th>vc-vc</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_1)</td>
<td>(b_2)</td>
<td>(s_1)</td>
<td>(s_2)</td>
<td>IE</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>7200</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>14400</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>5280</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>5</td>
<td>3</td>
<td>3750</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>3</td>
<td>5</td>
<td>1315*</td>
</tr>
<tr>
<td>5</td>
<td>36000</td>
<td>2</td>
<td>5</td>
<td>5760</td>
</tr>
<tr>
<td>225</td>
<td>7200</td>
<td>3</td>
<td>1</td>
<td>192*</td>
</tr>
<tr>
<td>21600</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>9600</td>
</tr>
</tbody>
</table>

*: predicted best., †: actual best.

the virtual cyclic to virtual cyclic scheme. Table 4, 5, and 6 give the indexing cost estimates and indexing times for the data sets (a), (b), and (c), respectively. For the last three cases in each of the data sets, either the source or the target distribution is chosen to be a block distribution.

Comparing the indexing estimates and the indexing times for the scheme with lowest indexing cost indicates that the selection criteria developed in Section 3.3 provide a good prediction of the best indexing scheme. In all the measured cases, the “predicted best” scheme has the lowest indexing time. Furthermore, the relative order among the indexing cost estimates is the same as that between the indexing times for the four schemes. For schemes with the same predicted cost, those having a virtual cyclic view for either the source or target distribution have a higher indexing time. This can be attributed to the fact that the non-unit stride between consecu-
tive elements in the data index sets for the virtual cyclic view results in poor cache performance due to a decrease in spatial locality.

Phase two uses tables generated during phase one of the virtual processor approach as shown in Fig. 24 and Fig. 25. Table 7, Table 8, and Table 9 gives the execution times and the table generation times for the cases shown in Table 4, Table 5, and Table 6, respectively. The execution times reported are for Phase two of the virtual processor approach and include the indexing time as well as the communication time. The best table generation and execution times are marked in Table 7, Table 8, and Table 9. Comparing the lowest indexing times in Table 4, Table 5, and Table 6 with the lowest execution times in Table 7, Table 8, and Table 9, respectively, it is observed that the virtualization scheme which has the lowest indexing cost also has the lowest execution time. This follows by noting that the communication pattern among the processors for all four schemes is identical. Thus the communication time for the four schemes is nearly equal and the scheme with lowest indexing time also has the lowest execution time. Comparing the lowest execution times and lowest table generation times in Table 7, Table 8, and Table 9, it is further observed that the scheme with the lowest execution time also has the lowest table generation time. This can be attributed to the fact that the table generation time is also directly proportional to the total number of sections generated. A comparison of the table generation time and the execution times for the best cases shows that for most of the considered cases, the table generation time is in the range of 5% to 50% of the execution time.
Table 7: Comparison of the Execution Time and the Table Generation Time \((P = 32, M = 230400)\).

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Execution Time (ms)</th>
<th>Table Generation Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vb-vb</td>
<td>vb-vc</td>
</tr>
<tr>
<td>(b_1)</td>
<td>(b_2)</td>
<td>(s_1)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>7200</td>
<td>1</td>
</tr>
<tr>
<td>225</td>
<td>7200</td>
<td>1</td>
</tr>
<tr>
<td>7200</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

*: best execution time, †: best table generation time.

3.6 Discussions

The developed scheme for compilation of the array statement can be extended to cover compilation of some forms of other data parallel constructs, such as the WHERE statement in Fortran 90 and for building the parameter list when passing array sections of distributed arrays to a subroutine call. We do not address the execution of statements with vector-valued subscripts which are allowed in the HPF FORALL statement and in the Fortran 90 array and WHERE statement. In general, runtime schemes as proposed in [113] will be required to handle vector valued subscripts.

The execution of the array statement in this chapter follows the owner-computes rule which is prevalent in compilers for HPF-like languages. However, the closed form expressions developed in Section 3.2 are not restricted to the owner computes-rule and
Table 8: Comparison of the Execution Time and the Table Generation Time ($P = 32, M = 230400$).

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Execution Time (ms)</th>
<th>Table Generation Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vb-vb</td>
<td>vb-vc</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$s_1$</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>21600</td>
<td>3</td>
</tr>
<tr>
<td>225</td>
<td>36000</td>
<td>5</td>
</tr>
<tr>
<td>14400</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

*: best execution time, †: best table generation time.

can be either directly used or adapted to cases where the iterations are distributed using a regular distribution as in KALI [82]. For instance, consider the KALI-like code in Fig. 28, in which the iterations of the specified loop and the arrays $A$ and $B$ are distributed among the processors using a block-cyclic distribution. The execution of the loop can be performed by the equivalent sequence of array statements in Fig. 29. The array $T(0 : N - 1)$ is distributed identically as the iterations of the loop. The communication sets for the array statements can be determined using the virtual processor approach and the closed form expressions developed in Section 3.2.

### 3.7 Summary

In this chapter, we have presented techniques for efficient enumeration of the local index sets, data index sets and processor sets for array statement execution involving
Table 9: Comparison of the Execution Time and the Table Generation Time ($P = 32, M = 230400$).

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Execution Time (ms)</th>
<th>Table Generation Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$b_1$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>225</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>36000</td>
<td>2</td>
</tr>
<tr>
<td>225</td>
<td>7200</td>
<td>3</td>
</tr>
<tr>
<td>21600</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

*: best execution time, †: best table generation time.

forall $i$ in 0..$N - 1$ on $(i \text{ div } b) \mod P$

$B(l_2 + is_2) = f(A(l_1 + is_1))$

endforall

Figure 28: Execution of program with explicit computation distribution.

arrays distributed using the block, cyclic, and block-cyclic distributions. Parameterized closed forms expressed as regular sections for determining the send and receive data and processor sets for block and cyclic distributions were derived. A strategy based on virtual processors was used along with these closed forms to generate efficient indexing code for arrays distributed using block-cyclic distributions. A heuristic for selecting the appropriate virtual views at the source and target distribution was
developed. Performance results for an implementation on the Cray T3D were presented. The performance results demonstrate that the developed heuristic provides a good indication of the virtualization scheme with the lowest execution time. It is also observed that for a majority of the cases considered the table generation overhead for the virtual processor approach is a small percentage of the execution time.

The virtual processor approach can be further optimized to handle frequently occurring cases such as data redistribution and array expressions with unit strides and/or identically distributed arrays, more efficiently. These optimizations will reduce the table generation time but are case-specific.
HPF supports a two-level mapping of data arrays to an abstract processor grid. The language introduces a Cartesian grid referred to as a *template*. Arrays are *aligned* to the template and the templates are *distributed* onto the abstract processor array using regular data distributions. The alignment of the array elements to the template elements can be performed at a stride with an initial offset. For instance, an array \( D(0 : N - 1) \) can be aligned with a template \( T \) such that \( D(i) \) is aligned with \( T(ci + a) \).

If memory for the local array is allocated for all the template cells mapped to the processor, then the introduction of a non-unit stride \( c \) in the array-template mapping creates "holes" in the array \( D.loc \) of each processor as some template cells do not have any array elements mapped to them. These holes represent memory which will never be utilized and hence should be removed by allocating memory only for those template cells with which array elements are aligned.

The indexing methods presented in Chapter III are directly applicable for an array statement involving arrays which are distributed using a two-level mapping when existence of holes in the local memory arrays is acceptable. Consider an array \( A(m_1 : n_1) \) aligned with template \( T_1 \) using a stride \( c_1 \) and an offset \( a_1 \) and an array
B(m_2 : n_2) aligned with template T_2 using a stride c_2 and an offset a_2. The array statement

\[ B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1)) \]  \hspace{1cm} (4.1)

is equivalent to the following array statement in terms of the sections of the templates T_1 and T_2:

\[ T_2(a_2 + l_2c_2 : a_2 + u_2c_2 : s_2c_2) = f(T_1(a_1 + l_1c_1 : a_1 + u_1c_1 : s_1c_1)) \]  \hspace{1cm} (4.2)

This follows by noting that the array elements in the section A(l_1 : u_1 : s_1) are aligned with the template cells in the template section T_1(a_1 + l_1c_1 : a_1 + u_1c_1 : s_1c_1) and those in the array section B(l_2 : u_2 : s_2) are aligned with the template cells in the template section T_2(a_2 + l_2c_2 : a_2 + u_2c_2 : s_2c_2). The methods developed in Chapter III, when applied to the array statement in Eq. 4.2, will construct the data index sets and local index sets in terms of the local indices of the distributed templates T_1 and T_2.

The hole compression would only affect how the array elements are placed in the local memory of a processor and would not affect which array elements are mapped to a processor. The latter depends only on the alignment of the array with the template and the distribution chosen for the template. Thus, the processor send and receive sets derived using the techniques for one-level mappings for the array statement in Eq. 4.2 are equal to those for the two-level mappings. Thus we have to concern ourselves only with the enumeration of the modified data and local index sets.

In this chapter, we will frequently be referring to two forms of allocation for the local array in a processor’s memory - one with hole compression and the other without.
Since the local array allocation without hole-compression is equivalent to allocating memory for the portion of the distributed template $T$ assigned to a processor, we refer to such a local array as $T.loc$. The local array allocation with hole-compression corresponds to storing only elements of the array $A$ and such a local array is referred to as $A.loc$. Since it is possible that multiple arrays are aligned with the same template, to distinguish between the corresponding uncompressed local arrays, we qualify $T.loc$ with the array name. For instance, if $A$ and $B$ are aligned with a template $T$, then the local arrays for $A$ and $B$ without hole compression are referred to as $T.loc.A$ and $T.loc.B$, respectively. The compression of holes in the local arrays complicates the indexing for the array statement. Specifically two problems have to be addressed:

- **Local Memory Allocation:** Determining the exact amount of the memory to be allocated for the distributed array in a processor's local memory. This problem is equivalent to enumerating the template cells mapped to the processor which have array elements aligned with them. Clearly, a goodness measure for efficiency of any local memory allocation scheme is the number of empty spaces remaining in the local array.

- **Index Translation:** The closed forms for data and local index sets developed in Chapter III are expressed in terms of indices of $T.loc$. After hole-compression the local indices of the array elements in $A.loc$ are different from those of the template cells they were aligned with in $T.loc$. The compression of holes in the local array modifies the local indices of the array elements. Hence new closed forms in terms of indices of $A.loc$ for enumerating the send and receive
data index sets and the local index set to be accessed for the execution of the array statement are to be developed. A goodness measure for the indexing scheme is the additional indexing overhead required compared to that required for enumerating the index sets for arrays distributed using one-level mappings.

To illustrate the above problems consider the array section \( A(0 : 36 : 3) \) of the array \( A(0 : 37) \) shown in Fig. 30. The distributed array is shown in Fig. 30(a) and the elements belonging to the array section \( A(0 : 36 : 3) \) are marked. Local memory allocation performed on basis of the template is shown in Fig. 30(b), while allocation with hole-compression performed is shown in Fig. 30(c). After hole compression, each processor will have a different number of array elements mapped to it and ideally a different amount of local memory should be allocated per processor. The local index set on processor 0 corresponding to the array section \( A(0 : 36 : 3) \) consists of the indices \{0, 6, 21, 27\} in \( T.loc \) and the indices \{0, 2, 7, 9\} in \( A.loc \). The methods developed in Chapter III, using the virtual processor approach would enumerate the set of indices \{0, 6, 21, 27\}. The objective of this chapter is to develop efficient methods for enumeration of the set of indices \{0, 2, 7, 9\}.

As will be observed in the following sections, a reduction in the amount of memory wasted comes at the added expense of greater indexing costs. The scheme of choice would depend on the relative importance of memory wastage reduction as compared to compile time and execution time reduction. For instance, if sufficient amount of memory is available, then no hole-compression need be performed leading to lower compile and execution times. However, if memory is at a shortage, then it is important
Figure 30: Memory allocation for array mapped using a two-level mapping with non-unit stride viewed under the virtual-cyclic view.
to compress holes and pay the additional cost of a higher execution time and compile time for the array statement.

Our approach for addressing the memory allocation and indexing problems for arrays mapped using two-level mappings with non-unit strides is based on the following observations:

1. For block and cyclically distributed templates the set of array elements mapped to a processor can be represented by a simple regular section.

2. The array indices represented by a simple regular section in the uncompressed array $T_{loc}$ can be represented by a simple regular section in the compressed array $A_{loc}$.

For block and cyclically distributed templates, the first observation facilitates the identification of the exact amount of memory to be allocated for each processor. It also helps construct a simple and computationally inexpensive scheme for mapping a template cell to which an array element is aligned to an element of the compressed array. Furthermore, for block and cyclically distributed templates, the data index sets and local index sets for the array statement in Eq. 4.2, can be expressed as simple regular sections of the uncompressed local arrays $T_{1,loc}$ and $T_{2,loc}$. It is this ability of representation in terms of regular sections, that facilitates the separation of the sending and receive phase (Chapter III, Section 3.3.2) into two phases to eliminate multiple messages between two processors. From the second observation, it follows that a similar scheme can be used for array statements involving arrays mapped using a two-level mapping with non-unit stride and a compressed local memory allocation.
For block and cyclically distributed templates, using the regular section characterization of the template cells mapped to a processor $p$, we develop a regular section characterization of the array elements located on $p$. This regular section characterization is used to determine the exact amount of memory to be allocated for the distributed array on processor $p$. Next, we develop a procedure for performing the translation between the local template section and the local array section. Using this procedure, the parameterized closed forms for the send and receive data index sets and the local index sets for one-level mappings can be extended to those for the two-level mappings. Using the results for the block and cyclically distributed arrays, the virtual processor approach for handling block-cyclic distributed templates is extended. The key idea in the extension is to look upon the local array as a two-dimensional array with one dimension corresponding to the virtual processor index and the other corresponding to the local index space of the virtual processor.

This chapter is organized as follows. In Section 4.1 we describe mathematical properties for regular sections which will be used in this chapter. A procedure for translating the data and local index sets of $T_{loc}$ to obtain the corresponding sections of the compressed array $A_{loc}$ are developed in Section 4.2. Extensions to the virtual processor approach for two-level mappings are provided in Section 4.3. Performance results on a Cray T3D are provided in Section 4.4.
4.1 Mathematical Preliminaries

The evaluation of the regular section characterization of the elements of an array located on a template uses several results about the intersection, translation, and expansion/compression of regular sections. We state the results without proof. Lemma 4.1 provides the result for computing the intersection of two regular sections. The result follows directly from the techniques for determining the solution of linear diophantine equations.

Lemma 4.1 (Regular Section Intersection) For any $c_1 > 0$ and $c_2 > 0$, we have

\begin{align*}
(a_1 : b_1) \cap (a_2 : b_2) &= (\max (a_1, a_2) : \min (b_1, b_2)) \quad (4.3) \\
(a_1 : b_1) \cap (a_2 : b_2 : c) &= (\max ((a_1 + (a_2 - a_1) \mod c), a_2) : \min (b_1, b_2, c)) \quad (4.4) \\
(a_1 : b_1 : c_1) \cap (a_2 : b_2 : c_2) &= \begin{cases} 
(\min (b_1, b_2), c) 
& \text{if } a_2 \equiv a_1 \mod \gcd(c_1, c_2) \quad (4.5) \\
\phi 
& \text{otherwise}
\end{cases} \\
\end{align*}

where $m$ is the array element such that $m = a_1 + ic_1 = a_2 + jc_2$ and $i$ is the smallest nonnegative integer for which $j$ is also nonnegative.

Let $u$ and $v$ be integers such that $c_1u - c_2v = \gcd(c_1, c_2)$. Identifying $m$, the first element of the intersection, corresponds to finding a particular solution pair $(i, j)$ for the linear diophantine equation $a_1 + c_1x = a_2 + c_2y$ in integer variables $x$ and $y$. The solution pair has the following characteristics: $i$ is the smallest nonnegative integer for which $j$ is also nonnegative. Suppose the linear diophantine equation $c_1x - c_2y = (a_2 - a_1)$ is solvable, i.e., $\gcd(c_1, c_2)|(a_2 - a_1)$. Since we know that
\( c_1 u - c_2 v = \gcd(c_1, c_2) \), the parameterized solutions for \( x \) and \( y \) in terms of the parameter \( t \) are as follows:

\[
\begin{align*}
\frac{x}{c_1} &= \frac{(a_2 - a_1)u}{\gcd(c_1, c_2)} + \frac{c_2 t}{\gcd(c_1, c_2)} \\
\frac{y}{c_2} &= \frac{(a_2 - a_1)v}{\gcd(c_1, c_2)} + \frac{c_1 t}{\gcd(c_1, c_2)}
\end{align*}
\]

(4.7) (4.8)

We are interested in finding a non-negative solution for \( x \), which implies that \( \frac{(a_2 - a_1)u}{\gcd(c_1, c_2)} + \frac{c_2 t}{\gcd(c_1, c_2)} > 0 \). Thus the smallest valid value for the parameter \( t \) is \( t = \left\lfloor \frac{(a_2 - a_1)u}{c_2} \right\rfloor \). The corresponding array section element is \( a_1 + \left\lfloor \frac{(a_2 - a_1)c_1 u}{c_2} \right\rfloor \). However, this element need not correspond to a non-negative value for variable \( y \). If \( m \) is the first element of the intersection then the next element of the intersection can be shown to be at a stride distance of \( \operatorname{lcm}(c_1, c_2) \). Thus the first element of the intersection is \( x \), such that \( x > \max(a_1, a_2) \) and \( x = a_1 + \left( \left\lfloor \frac{(a_2 - a_1)c_1 u}{c_2} \right\rfloor \right) \mod \lcm(c_1, c_2) \). Thus we have

\[
m = \max(a_1, a_2) + \left( \left( a_1 + \left\lfloor \frac{(a_2 - a_1)c_1 u}{c_2} \right\rfloor \right) - \max(a_1, a_2) \right) \mod \lcm(c_1, c_2) \quad (4.9)
\]

The function \( \text{first}(a_1, c_1, a_2, c_2) \) is defined to be the first element of the intersection of the sections \((a_1 : b_1 : c_1)\) and \((a_2 : b_2 : c_2)\).

Lemma 4.2 provides results for translating, expanding, and compressing regular sections. The section \(((l : u : s) \operatorname{op} c)\), where \( \operatorname{op} \in \{*, \div\} \), consists of the integers \(\{l + i * s) \operatorname{op} c \mid 0 \leq i \leq \left\lfloor \frac{u-l}{s} \right\rfloor\}\).

**Lemma 4.2** For a regular section \((l : u : s)\) and \(c \in \mathbb{Z}\) we have

[Translation] \((l : u : s) + c = (l + c : u + c : s)\) (4.10)

[Expansion] \((l : u : s) * c = (l * c : u * c : s * c)\) (4.11)

[Compression] \((l : u : s) \div c = (l \div c : u \div c : s \div c)\) if \(c | s \land c | l\) (4.12)
The above operations can be consecutively applied. For instance \((l : u : s) \ast a + b = (al + b : au + b : as)\). We now develop the regular section characterization of the array elements mapped to a processor for an array which is aligned with a template that is distributed among the processors using either a block or a cyclic distribution.

4.2 Block and Cyclically Distributed Templates

In this section, we address the memory allocation and indexing problem for block and cyclically distributed templates.

Consider an array \(A(m_1 : n_1)\) aligned with a template \(T(q_1 : r_1)\) at a stride \(c\) and an offset \(a\). Let the template \(T\) be distributed over \(P\) processors using either a block or a cyclic distribution. The elements of the template \(T\) assigned to a processor \(p\) due to the distribution can be represented by a regular section \(T(l_p : u_p : s_p)\), where

\[
\begin{align*}
l_p &= p \left(\left\lfloor \frac{r_1 - q_1 + 1}{P} \right\rfloor \right) \\
u_p &= \min\left( p \left(\left\lfloor \frac{r_1 - q_1 + 1}{P} \right\rfloor + \left\lfloor \frac{r_1 - q_1 + 1}{P} \right\rfloor - 1, r_1 \right) \right) \\
s_p &= 1
\end{align*}
\] (4.13) (4.14) (4.15)

if the template is distributed using a block distribution or

\[
\begin{align*}
l_p &= q_1 + p, \quad u_p = r_1, \quad s_p = P
\end{align*}
\] (4.16)

if the template is distributed using the cyclic distribution. We use \(T(l_p : u_p : s_p)\) as the regular section characterization of the template section located on the processor \(p\) to determine the regular section characterization of the elements of array \(A\) located on \(p\). We later instantiate the closed forms for \(l_p\), \(u_p\), and \(s_p\) depending on the
distribution of the template to obtain specific closed forms for either the block or the cyclic distribution.

4.2.1 Memory Allocation

Array $A(m_1 : n_1)$ is aligned with $T$ using an offset $a$ and a stride $c$. Thus the regular section $T(a + m_1c : a + n_1c : c)$ consists of all the template cells to which elements of $A$ are aligned. Since $T(l_p : u_p : s_p)$ consists of all the template cells located on processor $p$, all the template cells on processor $p$ which have array elements aligned with them are given by the intersection of these two regular sections, i.e.,

$$T(l_{pa} : u_{pa} : s_{pa}) = T(l_p : u_p : s_p) \cap T(a + m_1c : a + n_1c : c). \quad (4.17)$$

The intersection of regular sections can be computed using the result of Lemma 4.1. If the above intersection is empty then no elements of the array $A$ are located on processor $p$.

The array elements in the array section $A(l : u : s)$ are aligned with the template cells in the template section $T(a + lc : a + uc : sc)$. The inverse relation also holds. That is if $T(l : u : s)$ represents a section of the template $T$ such that an array element is aligned with every element of $T(l : u : s)$, then the array section corresponding to this template section is given by $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$. Note that it is necessary that $c$ the alignment stride divide $s$ the stride in the template section, else it is not possible that every element of $T(l : u : s)$ has an element of the array aligned with it. This translation is illustrated in Fig. 31 for array $A$ aligned with template $T$ at stride two and offset zero. The translation for the template section $T(0 : 14 : 4)$ is illustrated.
Figure 31: Functions for translating between template sections and array sections.

Now all the template cells on processor $p$ which have array elements aligned with them are included in the template section $T(l_{pa} : u_{pa} : s_{pa})$. Thus the array section $A \left( l_{pa} - a \cdot \frac{u_{pa} - a}{c} : s_{pa} \right)$ consists of all the array elements located on processor $p$. Thus we have obtained a regular section characterization of the elements of $A$ located on processor $p$ due to the two-level distribution of array $A$ onto the set of processors.

The memory allocation problem can now be solved in a straightforward fashion. Since the array section $A \left( l_{pa} - a \cdot \frac{u_{pa} - a}{c} : s_{pa} \right)$ consists of

$$N_{loc} = \frac{u_{pa} - a - l_{pa} - a}{c} \cdot \frac{s_{pa}}{c} \quad (4.18)$$

elements, it is sufficient to allocate the compressed local array on processor $p$ as $A_{loc}(0 : N_{loc})$. The array element $A(l_{pa} - a + i \frac{s_{pa}}{c})$ is mapped to the element $A_{loc}(i)$. Let $l_a = \frac{l_{pa} - a}{c}$, $u_a = \frac{u_{pa} - a}{c}$, and $s_a = \frac{s_{pa}}{c}$. We refer to $l_a$, $u_a$ and $s_a$ as the distribution parameters of the array $A$ on processor $p$. The values of the distribution parameters
Figure 32: Methodology for determining size of local array with hole compression.

are used in the index translation between a local template section and a local array section as shown in Section 4.2.2.

It is important to note that the above derivation used properties of translation, intersection, compression and expansion of regular sections. A schematic of the construction is shown in Fig. 32.
4.2.2 Index Set Translation

For block and cyclically distributed arrays, the data index sets and local index sets for the array statement in Eq. 4.2, can be expressed as simple regular sections of the uncompressed local array $T_{loc}$. We now show that a regular section in $T_{loc}$ corresponds to a regular section in $A_{loc}$ for block and cyclically distributed templates and develop a strategy for performing the translation. Fig. 33, illustrates the sequence
of transformations. Let $T.loc(l' : u' : s')$ be the local template section representing the data or local index set. Depending on the distribution of the template, $T.loc(l' : u' : s')$ will correspond to a global template $T(l : u : s)$. For a block distribution, we have:

$$
\begin{align*}
    l &= l' + q_1 + p \left\lfloor \frac{r_1 - q_1 + 1}{P} \right\rfloor, \\
    u &= u' + q_1 + p \left\lfloor \frac{r_1 - q_1 + 1}{P} \right\rfloor, \\
    s &= s'.
\end{align*}
$$

For the cyclically distributed template we have

$$
\begin{align*}
    l &= q_1 + p + l'P, \\
    u &= q_1 + p + u'P, \\
    s &= s'P.
\end{align*}
$$

Furthermore, the techniques developed in Chapter III guarantee that every cell in the template section $T(l : u : s)$ has an array element aligned with it. Thus the section of array elements represented by the template cells is $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$. As shown in Section 4.2.1, the array section $A \left( l_a : u_a : s_a \right)$ consists of all the elements of array $A$ located on processor $p$. Thus the array section $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$ must be a subsection of this section as shown in Fig. 34. The first element of $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$ has the same global index as $A(l_a + ic_a)$, where $i = \frac{l-a}{c_a} - \frac{l_a}{c_a}$. Since $A(l_a + ic_a)$ is mapped to $A.loc(i)$, the first element of the section $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$ will map to $A.loc \left( \frac{l-a}{c_a} \right)$. The stride between consecutive elements of $A \left( \frac{l-a}{c} : \frac{u-a}{c} : \frac{s}{c} \right)$ in $A.loc$ is $\frac{s}{c_a} = \frac{s}{c_a c}$. Thus the local template section, $T.loc(l' : u' : s')$ will map to the following section of
Thus once the distribution parameters \( l_a \) and \( c_a \) for the distributed array \( A \) have been evaluated, any data index set or local index set in \( T.loc \) can be translated to obtain the corresponding index set in \( A.loc \).

We now instantiate the above formulas to obtain specific closed forms for block and cyclically distributed templates. These closed forms can be directly used for the translation of the index sets.

### 4.2.3 Block Distributed Template

Suppose the template \( T(q_1 : r_1) \) is distributed on \( P \) processors using a block distribution. The template section \( T(l_p : \min(l_p + B - 1, r)) \), where \( l_p = q_1 + pB \) and...
\( B = \lceil \frac{p - q + 1}{p} \rceil \), represents the global indices of the template cells mapped to processor \( p \). From Lemma 4.1, the global indices of the template cells mapped to processor \( p \) and having array elements aligned with them are given by the intersection:

\[
T(l_p : l_p + B - 1) \cap T(a + m_1 c : a + n_1 c : c) = T(\max(l_p + (a + m_1 c - l_p) \mod c, a + m_1 c) : \min(l_p + B - 1, a + n_1 c) : c) \quad (4.26)
\]

Thus the distribution parameters on processor \( p \) for an array \( A(m_1 : n_1) \) aligned with template \( T \) are given by:

\[
\begin{align*}
l_a &= \max(l_p + (a + m_1 c - l_p) \mod c, a + m_1 c) - a \\
u_a &= \min(l_p + B - 1, a + n_1 c) - a \\
c_a &= 1
\end{align*} \quad (4.27)
\]

The size of the local array is \( N_{loc} = u_a - l_a + 1 \).

Consider the section \( T_{loc}(l' : u' : s') \). For the block distribution, the corresponding section in template global space is \( T(l'' + q_1 + pB_1 : u' + q_1 + pB_1 : s') \), which can be translated to obtain the corresponding section of array elements \( A\left(\frac{l'' + q_1 + pB_1 - a}{c} : \frac{u' + q_1 + pB_1 - a}{c} : \frac{s'}{c}\right) \). Finally, using the distribution parameters \( l_a \) and \( c_a \), the section corresponding to \( T_{loc}(l' : u' : s') \) in \( A_{loc} \) is determined to be:

\[
T_{loc}(l' : u' : s') \to A_{loc}\left(\frac{l'' + q_1 + pB_1 - a}{c} - l_a : \frac{u' + q_1 + pB_1 - a}{c} - l_a : \frac{s'}{c}\right) \quad (4.28)
\]

Eq. 4.31 provide the requisite index-set translation functions for a block distributed template.
4.2.4 Cyclically Distributed Template

Suppose the template \( T(q_1 : r_1) \) is distributed on \( P \) processors using a cyclic distribution. The template section \( T(q_1 + p : r_1 : P) \), represents the global indices of the template cells mapped to processor \( p \). From Lemma 4.1, the global indices of the template cells mapped to processor \( p \) and having array elements aligned with them are given by the intersection:

\[
T(q_1 + p : r_1 : P) \cap T(a + m_1c : a + n_1c : c) = \begin{cases} 
T(m : \min(r_1 : a + n_1c) : \lcm(c, P)) & \text{if } g|(a + m_1c - q_1 - p) \\
0 & \text{if } g \nmid (a + m_1c - q_1 - p)
\end{cases}
\] (4.32)

where \( g = \gcd(P, c) \), and \( m = \text{first}(q_1 + p, P, a + m_1c, c) \) as defined in Lemma 4.1. Thus the distribution parameters on processor \( p \) for an array \( A(m_1 : n_1) \) aligned with template \( T \) are given by

\[
\begin{align*}
    l_a &= \frac{\text{first}(q_1 + p, P, a + m_1c, c) - a}{c} \\
    u_a &= \frac{\min(r_1, a + n_1c) - a}{c} \\
    c_a &= \frac{\lcm(c, P)}{c} = \frac{P}{\gcd(c, P)}
\end{align*}
\]

(4.33) (4.34) (4.35)

The size of the local array is \( N_{loc} = \frac{u_a - l_a}{c_a} + 1 \).

Consider the section \( T.loc(l' : u' : s') \). For the cyclic distribution, the corresponding section in template global space is \( T(q_1 + p + l'P : q_1 + p + u'P : s'P) \), which can be translated to obtain the corresponding section of array elements \( A \left( \frac{q_1 + p + l'P - a}{c} : \frac{q_1 + p + u'P - a}{c} : \frac{s'P}{c} \right) \). Finally, using the distribution parameters \( l_a \) and \( c_a \),
the section corresponding to $T_{loc}(l' : u' : s')$ in $A_{loc}$ is determined to be:

$$T_{loc}(l' : u' : s') \rightarrow A_{loc} \left( \frac{v_{1} + p + t' P - a}{\text{lcm}(c, P) / c} : \frac{v_{1} + p + u' P - a}{\text{lcm}(c, P) / c} : \frac{s' P}{c_{a,c}} \right) \quad (4.36)$$

Eq. 4.36 provide the requisite index-set translation functions for a block distributed template.

### 4.3 Virtual Processor Approach for Two-Level Mappings

We now extend the memory allocation scheme and the index translation scheme to block-cyclically distributed templates using the virtual processor approach. The virtual processor approach is based on viewing a block-cyclic distribution as a block (or cyclic) distribution on a set of virtual processors, which are cyclically (or block-wise) mapped to the physical processors. The closed forms developed for the block and cyclically distributed arrays, are used for computing the data and local index sets in the local index space of the virtual processors, and then extended to the local index space of the physical processor to which the virtual processor is mapped. For one-level mappings, this translation between the local index spaces of the virtual processor and physical processor can be performed in a straightforward fashion. However, for arrays mapped using a two-level mapping with non-unit stride and local memory allocation performed only for array elements, this translation is not straightforward.

Under the virtual block view, the block-cyclic distribution is viewed as a block distribution in the virtual processor domain, i.e., one block of data is owned by each virtual processor. The virtual processors are then cyclically distributed among the physical processors. Thus the total number of elements per virtual processor, except
possibly the last virtual processor, is identical for all the virtual processors. The offset of the origin of the index space of a virtual processor \( v \) into the index space of a physical processor \( p \) to which it is mapped is equal to the number of virtual processors mapped to \( p \) with index lower than \( v \), times the number of array elements per virtual processor. The stride in the virtual processor index space does not change in the physical processor index space. For two-level mappings with hole-compression, this translation is not straightforward, as the number of array elements per virtual processor need not be identical for all the virtual processors. For instance, consider the two-level mapping of the array \( A(0:37) \) at a stride two and offset zero (Fig. 30). Under the virtual block view, each virtual processor will have either one or two array elements in its local space and determining the offset of the index space of a virtual processor into that of the physical processor is more complicated. Hence, while the results developed in the previous section, for memory allocation and index translation for block and cyclically distributed templates can be directly applied to obtain the corresponding index sets in the compressed local array of each virtual processor, a scheme for translating these sections to the compressed index space of the physical processors is required.

A similar problem arises for the virtual-cyclic view. Under the virtual cyclic view, a block-cyclic distribution is viewed as a cyclic distribution on a set of virtual processors which are distributed in a block-wise fashion onto the physical processors. Thus the origin of the index space of a virtual processor \( v \) in the index space of the physical processor \( p \) to which it is mapped is fixed at \( v \div b \) for a \textit{cyclic}(b)
Figure 35: Memory allocation for array mapped using a two-level mapping with non-unit stride.
distribution. Furthermore, a unit stride between the elements in the virtual processor index space, corresponds to a fixed stride $b$ in the physical processor index space where $b$ is the block size of the block-cyclic distribution. Thus for one-level mappings, a regular section in the virtual processor local index space can be translated to the physical processor local index space in a straightforward fashion. Under the two-level mapping with hole compression, each virtual processor may not have the same number of elements or any elements at all. Hence it is possible that both the origin and the stride of the virtual processor’s index space in the physical processor’s space may change. For instance, consider the array $A(0 : 15)$, aligned with a template $T(0 : 31)$ at a stride two and offset zero. The template is distributed among two processors using a cyclic(4) distribution. Fig. 35(a) shows the distribution of the array among the processors, Fig. 35(b) illustrates the local memory allocation without hole-compression, and Fig. 35(c) illustrates the local memory allocation with hole compression. The virtual processors under the virtual-cyclic view are also marked in Fig. 35(a). All the virtual processors with odd indices do not have any array element in their local index spaces. In the compressed array, the origins of the local index spaces of the even-indexed virtual processors are modified.

Our approach for addressing these problems is based on allocating the local array as a two-dimensional array, wherein the virtual processor index varies along one dimension and the index within the local index space of the virtual processor varies along the other dimension. We now describe the details of memory allocation and indexing for the two virtual views.
4.3.1 Virtual Block View

Consider an array $A(m_1 : n_1)$ aligned with a template $T(q_1 : r_1)$ at a stride $c$ and an offset $a$. The template is distributed using a cyclic distribution onto $P$ processors. Under the virtual block view, the array is assumed to be block distributed on $VP = \max \left(\left\lfloor \frac{r_1-q_1+1}{b} \right\rfloor, 0 \right)$ virtual processors. The compressed local array is allocated such that the local index within the virtual processor index space varies along one dimension, while the virtual processor index varies along the other. The local memory allocation for each virtual processor is performed such that memory is allocated only for those template cells which have array elements aligned with them. The techniques developed in Section 4.2.3 can be used for performing the allocation. For instance, the two-dimensional local memory allocation for the array $A(0 : 37)$ (Fig. 30), is illustrated in Fig. 36. The two-dimensional allocation corresponds to performing a hole compression within a block. Since each virtual processor may not have equal number of array elements and since Fortran allows memory allocation for only rectangular arrays, some holes may still exist in the local array. These holes can be eliminated if a separate allocation for the exact amount of memory is performed for each virtual processor as opposed to allocating a two-dimensional array for the entire local array.

Under the virtual block view each block of template cells corresponds to a virtual processor. Due to the stride and offset in the alignment, it is possible that virtual processors mapped to a physical processor do not have any array elements mapped to them. Clearly no memory should be allocated for these virtual processors. Depending on the relationship between the alignment stride and the block size, it is possible to
Figure 36: Two-dimensional Local Memory Allocation for array mapped using a two-level mapping with non-unit stride under virtual block view.
allocate memory only for the virtual processors which have array elements mapped to them. If \( c \leq b \), then each virtual processor \( v \in ((cm_1 + a) \text{ div } b : (cn_1 + a) \text{ div } b) \) has at least one element of the array mapped to it. Let \( v_l = (cm_1 + a) \text{ div } b \) and \( v_u = (cn_1 + a) \text{ div } b \). Since the set of virtual processors on processor \( p \) is \((p : VP : P)\), the virtual processors on \( p \) with array elements allocated to them are

\[
(v_l : v_u) \cap (p : VP : P) = (\max(v_l + (p - v_l) \text{ mod } P, p) : \min(v_u, VP) : P).
\] (4.37)

Thus we allocate memory for these virtual processors and use the techniques from Section 4.2.3 to determine the allocation for each virtual processor. If \( c > b \), then each virtual processor has at most one element each and we allocate memory for every virtual processor.

Given a template section, \( T(l : u : s) \) the characterization of the active virtual processors on a processor, can be performed using techniques similar to those presented in Section 3.3.1. Depending on the set of virtual processor for which memory has been allocated, some translation of this set of active virtual processors will have to be performed. We now describe the allocation scheme for the virtual-cyclic view.

### 4.3.2 Virtual Cyclic View

Consider array \( A(m_1 : n_1) \) aligned with a template \( T(q_1 : r_1) \) at a stride \( c \) and an offset \( a \). The template \( T \) is distributed using a cyclic\((b)\) distribution. Under a virtual-cyclic view, the template is assumed to have a cyclic distribution on \( VP = \min(P * b, r_1 - q_1 + 1) \) virtual processors, which are block-wise distributed to \( P \) processors. The local array is allocated such that dimension zero corresponds to the local index.
within virtual processor index space, while dimension one corresponds to the virtual processor index. For instance the two-dimensional local memory allocation for the array \(A(0:37)\)(Fig. 30), is illustrated in Fig. 37.

Note that the storage for the virtual cyclic view corresponds to an alternative allocation strategy. If the two-dimensional array is stored using a row-major order then the order of storage of elements for the virtual-cyclic view will be different from that for the virtual-block view. It is possible that there exist many references to an array within a program. Depending on the array section referenced a different virtual view may have better performance for each section reference. However, a conversion between virtual views would require a conversion between the two local memory storage orders. We assume that a single virtual view is used throughout the course of the program and a performance penalty is paid at the points where the fixed virtual view is not the best virtual view.

Similar to the virtual block view, it is possible that some virtual processors under the virtual cyclic view may not have any array elements mapped to them. The template cells located on a virtual processor \(v\) have indices \((q_1 + v : r_1 : P * b)\). Thus if \(v\) has any array elements mapped to it, then the intersection \((q_1 + v : r_1 : P * b) \cap (a + m_1 c : a + n_1 c : c)\) is not empty. Hence a virtual processor \(v\) has array elements mapped to it if \(\gcd(c, P * b)| (q_1 + v - a - m_1 c)\). The first virtual processor \(v_f\) on a processor \(p\) can be found by noting that the \(v_f \in (p * b : p * b + b - 1)\) and \(\gcd(c, P * b)|(q_1 + v - a - m_1 c)\). Thus we have

\[
v_f = p * b + ((-p * b - q_1 + a + m_1 c) \mod \gcd(c, P * b))
\] (4.38)
Figure 37: Two-dimensional Local Memory Allocation for array mapped using a two-level mapping with non-unit stride under the virtual cyclic view.
Figure 38: Memory allocation for array mapped using a two-level mapping with non-unit stride.

The set of virtual processors on \( p \) to which array elements are possibly mapped is thus given by the regular section

\[
(v_f : \min(p \cdot b + b - 1, a + n_1 c - q_1) : \gcd(c, P \cdot b))
\]

Local memory allocation is performed only for these virtual processors. Application of the above method to the array in Fig. 35, identifies the set of virtual processors to which array elements are mapped as \((0 : 3 : 2)\) on processor 0 and \((4 : 7 : 2)\) on processor 1. Thus the allocation will be as shown in Fig. 38. The values of \( v_f \) and \( \gcd(c, P \cdot b) \) are thus required for performing the indexing along the virtual processor dimension.
Given a template section, \( T(\tilde{l} : \tilde{u} : c * s) \) the characterization of the active virtual processors on a processor, i.e., the set of virtual processors to which elements of the template section are mapped, can be performed using techniques similar to those presented in Section 3.3.1. Depending on the set of virtual processors for which memory has been allocated, some translation and compression of this set of active virtual processors will need to be performed. For instance, the set of virtual processors located on a processor \( p \) as given in Eq. 4.39 is the regular section \((v_f : \min(p * b + b - 1, a + n_1 c - q_1) : \gcd(c, P * b))\). Now as shown in Section 3.3.1, let the set of active virtual processors corresponding to the template section \( T(\tilde{l} : \tilde{u} : c * s) \) be the regular section \((\tilde{v}_f : \tilde{v}_u : \gcd(c * s, P * b))\). Since \((\tilde{v}_f : \tilde{v}_u : \gcd(c * s, P * b))\) is a subsection of \((v_f : \min(p * b + b - 1, a + n_1 c - q_1) : \gcd(c, P * b))\) the indexing into the local virtual processor array can be determined to be

\[
\left( \frac{\tilde{v}_f - v_f}{\gcd(c, P * b)} : \frac{\tilde{v}_u - v_f}{\gcd(c, P * b)} : \frac{\gcd(c * s, P * b)}{\gcd(c, P * b)} \right)
\]

We now evaluate the potential merits of the local memory allocation scheme with hole compression.

### 4.3.3 Strategy for Selection of Virtualization Schemes

The strategy for selection of the virtualization scheme for array statements involving arrays mapped using two-level mappings with non-unit stride and memory allocation with hole compression, is similar to that described in Section 3.3.3. Since the processor send and receive sets do not change due to the compression of holes, the estimates presented in Table 3 are directly applicable to the array statement in template index
space as shown in Eq. 4.2. Also the estimates for $V_{Act\_A_{max}}$ and $V_{Act\_B_{max}}$ presented in Eq. 3.60 and Eq. 3.61 are directly applicable. Based on the developed cost estimates a selection of the virtual view at the source and the target array can be performed.

4.4 Performance Results

In this section, we present experimental results to evaluate the performance of the virtual processor approach for handling arrays mapped using a two-level mapping with non-unit stride and for the memory allocation scheme with hole compression. The experiments were performed on a 32-node Cray T3D. The time required by each processor to execute the node program for an array assignment statement was measured and the maximum time among all processors reported. Times were measured using the rtclock() wall clock timer.

The goals of the performance measurement were to determine how the local memory allocation and indexing strategy with hole-compression performed with respect to that without hole compression. The basis for the performance comparison were the following performance metrics:

- **Table generation time**: the time for the generation of the index sets for the execution of the array statement. The table generation time is either a compile-time or a run-time cost depending on whether or not all the parameters for the array statement are known at compile-time.
- **Memory allocation time:** While memory allocation for the storage scheme without hole compression is straightforward, memory allocation for the indexing scheme with hole compression involves some additional computation at allocation time. This computation involves identifying the virtual processors for which memory should be allocated and computing the distribution parameters for these virtual processors. The memory allocation time is a compile-time cost for static arrays but would be performed at compile time for allocatable arrays.

- **Memory wastage:** The amount of memory which would be allocated under the two schemes but would never be used. This is the number of holes introduced due to allocating memory for template cells which do not have any array elements aligned with them.

- **Array statement execution time:** The time for the execution of the array statement. The modification of the storage pattern in the local array due to hole-compression leads to a change in the spatial locality characteristics for the array statement execution.

Hole compression leads to a modification of only the local index sets and send and receive data index sets. The processor sets for the execution of the array statement are unchanged. Hence, to evaluate the performance of the hole-compression technique we consider an array statement which would lead to no communication and focus only on the enumeration of the local index sets. The conclusions derived from the performance results for the local index sets, would be equally applicable to the schemes for enumerating the send and receive data index sets. To reduce the number of
independent parameters, we measured various times for the execution of an array assignment statement of the form:

\[ A(0 : M * s - 1 : s) = x * A(0 : M * s - 1 : s) + y. \]  \hspace{1cm} (4.41)

The numbers \( x \) and \( y \) are double precision floating point numbers (eight bytes). The double precision floating point array \( A \) is aligned with a template \( T \) with offset zero \((a = 0)\) and a stride \( c > 0 \). The template \( T \) is distributed using a \textit{cyclic(b)} distribution. Thus the equivalent array statement in the template index space is

\[ T(0 : M * c * s - 1 : c * s) = x * T(0 : M * c * s - 1 : c * s) + y. \]  \hspace{1cm} (4.42)

The parameter \( M \) gives the number of elements in the array section. For our experiments, we kept the number of active elements \( M \) fixed at 23040 and \( P = 32 \) processors were used. The varying parameters were: block size \( b \), alignment stride \( c \), and the array statement stride \( s \). The empirical values for the memory allocation time, the table generation time, memory wastage, and array statement execution time were obtained for various values of \( b \), \( c \), and \( s \). The values of the block size \( b \) considered were equally spaced between the block \((b = \lfloor \frac{M * c * s}{P} \rfloor)\) and the cyclic \((b = 1)\) distribution. All the reported times are in microseconds. We first present the table generation times.

Tables 10 to 15 present the table generation times for different block sizes and virtual views. For the block and the cyclic distributions only timings for the virtual block view and virtual cyclic view, respectively, are provided. For all the data points considered it can be observed that the table generation time for the indexing scheme with
Table 10: Comparison of table generation times for block distributed template under a virtual block view. $P = 32, M = 23040.$

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th></th>
<th>$c = 3$</th>
<th></th>
<th>$c = 5$</th>
<th></th>
<th>$c = 6$</th>
<th></th>
<th>$c = 8$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>14</td>
<td>5</td>
<td>14</td>
<td>5</td>
<td>14</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

Table 11: Comparison of table generation times for indexing schemes for cyclic(720) distributed template using virtual block view. $P = 32, M = 23040.$

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th></th>
<th>$c = 3$</th>
<th></th>
<th>$c = 5$</th>
<th></th>
<th>$c = 6$</th>
<th></th>
<th>$c = 8$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>17</td>
<td>9</td>
<td>28</td>
<td>14</td>
<td>41</td>
<td>15</td>
<td>49</td>
<td>19</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>30</td>
<td>15</td>
<td>50</td>
<td>23</td>
<td>79</td>
<td>27</td>
<td>96</td>
<td>34</td>
<td>124</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>42</td>
<td>21</td>
<td>72</td>
<td>33</td>
<td>116</td>
<td>39</td>
<td>138</td>
<td>50</td>
<td>184</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>57</td>
<td>27</td>
<td>97</td>
<td>43</td>
<td>154</td>
<td>50</td>
<td>186</td>
<td>66</td>
<td>251</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>70</td>
<td>33</td>
<td>116</td>
<td>52</td>
<td>189</td>
<td>62</td>
<td>228</td>
<td>82</td>
<td>299</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>82</td>
<td>39</td>
<td>138</td>
<td>63</td>
<td>231</td>
<td>74</td>
<td>269</td>
<td>98</td>
<td>357</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
<td>103</td>
<td>51</td>
<td>186</td>
<td>82</td>
<td>301</td>
<td>98</td>
<td>369</td>
<td>129</td>
<td>477</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression
Table 12: Comparison of table generation times for indexing schemes for \textit{cyclic}(80) distributed template using virtual block view. \( P = 32, M = 23040 \).

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>100</td>
<td>57</td>
<td>196</td>
<td>92</td>
</tr>
<tr>
<td>2</td>
<td>74</td>
<td>245</td>
<td>109</td>
<td>404</td>
<td>180</td>
</tr>
<tr>
<td>3</td>
<td>109</td>
<td>361</td>
<td>163</td>
<td>607</td>
<td>268</td>
</tr>
<tr>
<td>4</td>
<td>145</td>
<td>451</td>
<td>215</td>
<td>808</td>
<td>357</td>
</tr>
<tr>
<td>5</td>
<td>180</td>
<td>601</td>
<td>268</td>
<td>1013</td>
<td>445</td>
</tr>
<tr>
<td>6</td>
<td>216</td>
<td>696</td>
<td>322</td>
<td>1199</td>
<td>533</td>
</tr>
<tr>
<td>8</td>
<td>286</td>
<td>947</td>
<td>427</td>
<td>1621</td>
<td>710</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

Table 13: Comparison of table generation times for indexing schemes for \textit{cyclic}(720) distributed template using virtual cyclic view. \( P = 32, M = 23040 \).

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>2794</td>
<td>3089</td>
<td>2181</td>
<td>2696</td>
<td>1283</td>
</tr>
<tr>
<td>2</td>
<td>1599</td>
<td>1940</td>
<td>1096</td>
<td>1438</td>
<td>646</td>
</tr>
<tr>
<td>3</td>
<td>1095</td>
<td>1276</td>
<td>730</td>
<td>955</td>
<td>431</td>
</tr>
<tr>
<td>4</td>
<td>801</td>
<td>921</td>
<td>534</td>
<td>694</td>
<td>323</td>
</tr>
<tr>
<td>5</td>
<td>646</td>
<td>781</td>
<td>431</td>
<td>574</td>
<td>2584</td>
</tr>
<tr>
<td>6</td>
<td>534</td>
<td>621</td>
<td>367</td>
<td>470</td>
<td>218</td>
</tr>
<tr>
<td>8</td>
<td>401</td>
<td>461</td>
<td>272</td>
<td>352</td>
<td>164</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression
Table 14: Comparison of table generation times for indexing schemes for cyclic distributed template using virtual cyclic view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>304</td>
<td>359</td>
<td>147</td>
<td>182</td>
<td>553</td>
</tr>
<tr>
<td>2</td>
<td>180</td>
<td>208</td>
<td>77</td>
<td>99</td>
<td>260</td>
</tr>
<tr>
<td>3</td>
<td>553</td>
<td>630</td>
<td>1086</td>
<td>1323</td>
<td>252</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
<td>108</td>
<td>260</td>
<td>314</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>77</td>
<td>90</td>
<td>253</td>
<td>290</td>
<td>253</td>
</tr>
<tr>
<td>6</td>
<td>260</td>
<td>306</td>
<td>558</td>
<td>685</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>49</td>
<td>58</td>
<td>161</td>
<td>192</td>
<td>25</td>
</tr>
</tbody>
</table>

*Non* - without hole compression, *Com* - with hole compression

Table 15: Comparison of table generation times for indexing schemes for cyclic distributed template using virtual cyclic view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>17</td>
<td>17</td>
<td>26</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>35</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>17</td>
<td>24</td>
<td>31</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>23</td>
<td>20</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>28</td>
<td>25</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>18</td>
<td>20</td>
<td>28</td>
<td>21</td>
</tr>
</tbody>
</table>

*Non* - without hole compression, *Com* - with hole compression
hole-compression is greater than that for the scheme without hole-compression. This behavior is expected as the indexing scheme with hole compression has to perform an addition index translation from the local template index space to the compressed local array index space for every virtual processor which owns elements of the array section. The table generation times for the block distributed template (Table 10) is nearly independent of the alignment and array section strides. This follows by noting that under the virtual block view for a block distribution, a processor has one virtual processor mapped to it and the computation for the evaluation of the index set is nearly fixed. A similar behaviour would be expected for the cyclically distributed template under a virtual cyclic view (Table 15). But variations in the time for evaluation of \( \gcd(P \ast b, c \ast s) \) lead to the variations in table generation time. For the block-cyclic distributions under the virtual block view (Table 11 and Table 12), the table generation time increases with increasing array section stride \( s \) and alignment stride \( c \). This follows by noting that the template is \( T(0 : M \ast c \ast s - 1) \) and under the virtual block view, a block of size \( b \) belongs to a virtual processor. Thus for a fixed block size \( b \), the number of virtual processors mapped to a processor increases with increasing \( s \) and \( c \) and the table generation overhead increases with increasing number of virtual processors. For the block-cyclic distribution with \( b = 720 \), viewed under a virtual cyclic view (Table 13), the table generation time decreases with increasing \( s \) and \( c \) except for the data point corresponding to \( c = 5, s = 5 \). This follows by noting that the template section has a stride of \( c \ast s \) and the stride between consecutive active virtual processors is \( g = \gcd(P \ast b, c \ast s) \). For \( b = 720 \), and for the values of \( c \)
and $s$ considered, except for $c = 5$, $s = 5$, we have $gcd(P \times b, c \times s) = c \times s$. Thus the number of active virtual processors decreases with increasing $c$ and $s$ except for the point $c = 5$, $s = 5$. For $b = 80$ (Table 14), no regular pattern can be identified as $gcd(P \times b, c \times s)$ does not vary in a regular fashion with either $c$ or $s$. The values of the table generation time vary inversely with the value of $gcd(P \times b, c \times s)$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>91</td>
<td>58</td>
<td>150</td>
<td>58</td>
<td>184</td>
</tr>
<tr>
<td>2</td>
<td>183</td>
<td>91</td>
<td>184</td>
<td>91</td>
<td>184</td>
</tr>
<tr>
<td>3</td>
<td>184</td>
<td>150</td>
<td>184</td>
<td>150</td>
<td>184</td>
</tr>
<tr>
<td>4</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>184</td>
</tr>
<tr>
<td>5</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>185</td>
</tr>
<tr>
<td>6</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>184</td>
<td>185</td>
</tr>
<tr>
<td>8</td>
<td>184</td>
<td>182</td>
<td>183</td>
<td>182</td>
<td>184</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

We now present the execution times for the array statement. Tables 16 to 21, present the execution times for different block sizes and virtual views. The Cray T3D has a direct mapped data cache with 256 lines of size 32 bytes each. Thus four double floating point numbers should fit into a single cache line and spatial locality will be exploited when the effective access stride is less than four. For the virtual block view (Table 16, Table 17, and Table 18) and for the data points at which the stride $s$ is less than four the indexing scheme with hole compression performs better than
Table 17: Comparison of execution times for indexing schemes for cyclic(720) distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>92 60</td>
<td>155 58</td>
<td>193 60</td>
<td>195 60</td>
<td>194 64</td>
</tr>
<tr>
<td>2</td>
<td>191 95</td>
<td>195 98</td>
<td>201 103</td>
<td>203 106</td>
<td>205 112</td>
</tr>
<tr>
<td>3</td>
<td>194 157</td>
<td>201 161</td>
<td>208 170</td>
<td>213 172</td>
<td>220 182</td>
</tr>
<tr>
<td>4</td>
<td>194 197</td>
<td>203 202</td>
<td>218 215</td>
<td>220 218</td>
<td>234 235</td>
</tr>
<tr>
<td>5</td>
<td>201 199</td>
<td>208 208</td>
<td>223 222</td>
<td>235 226</td>
<td>244 241</td>
</tr>
<tr>
<td>6</td>
<td>203 202</td>
<td>213 206</td>
<td>235 224</td>
<td>242 242</td>
<td>251 252</td>
</tr>
<tr>
<td>8</td>
<td>205 208</td>
<td>220 210</td>
<td>245 245</td>
<td>252 257</td>
<td>286 292</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

Table 18: Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>110 69</td>
<td>190 77</td>
<td>254 91</td>
<td>264 111</td>
<td>292 132</td>
</tr>
<tr>
<td>2</td>
<td>241 131</td>
<td>263 163</td>
<td>327 185</td>
<td>345 221</td>
<td>386 267</td>
</tr>
<tr>
<td>3</td>
<td>263 230</td>
<td>308 284</td>
<td>394 363</td>
<td>438 389</td>
<td>498 473</td>
</tr>
<tr>
<td>4</td>
<td>292 206</td>
<td>344 347</td>
<td>463 459</td>
<td>499 508</td>
<td>630 642</td>
</tr>
<tr>
<td>5</td>
<td>326 323</td>
<td>394 383</td>
<td>537 529</td>
<td>611 583</td>
<td>723 728</td>
</tr>
<tr>
<td>6</td>
<td>345 351</td>
<td>438 420</td>
<td>611 584</td>
<td>678 704</td>
<td>743 780</td>
</tr>
<tr>
<td>8</td>
<td>386 400</td>
<td>499 523</td>
<td>723 745</td>
<td>743 762</td>
<td>1182 1173</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression
Table 19: Comparison of execution times for indexing schemes for cyclic(720) distributed template using virtual cyclic view. \( P = 32, M = 23040. \)

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>1107</td>
<td>500</td>
<td>853</td>
<td>341</td>
<td>630</td>
</tr>
<tr>
<td>2</td>
<td>723</td>
<td>345</td>
<td>562</td>
<td>258</td>
<td>422</td>
</tr>
<tr>
<td>3</td>
<td>562</td>
<td>293</td>
<td>429</td>
<td>179</td>
<td>352</td>
</tr>
<tr>
<td>4</td>
<td>440</td>
<td>197</td>
<td>374</td>
<td>150</td>
<td>308</td>
</tr>
<tr>
<td>5</td>
<td>422</td>
<td>174</td>
<td>352</td>
<td>130</td>
<td>671</td>
</tr>
<tr>
<td>6</td>
<td>374</td>
<td>152</td>
<td>312</td>
<td>126</td>
<td>272</td>
</tr>
<tr>
<td>8</td>
<td>328</td>
<td>117</td>
<td>284</td>
<td>96</td>
<td>250</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

Table 20: Comparison of execution times for indexing schemes for cyclic(80) distributed template using virtual cyclic view. \( P = 32, M = 23040. \)

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>209</td>
<td>91</td>
<td>384</td>
<td>156</td>
<td>234</td>
</tr>
<tr>
<td>2</td>
<td>240</td>
<td>88</td>
<td>306</td>
<td>110</td>
<td>208</td>
</tr>
<tr>
<td>3</td>
<td>304</td>
<td>205</td>
<td>401</td>
<td>279</td>
<td>235</td>
</tr>
<tr>
<td>4</td>
<td>213</td>
<td>67</td>
<td>241</td>
<td>84</td>
<td>194</td>
</tr>
<tr>
<td>5</td>
<td>208</td>
<td>81</td>
<td>235</td>
<td>94</td>
<td>235</td>
</tr>
<tr>
<td>6</td>
<td>242</td>
<td>183</td>
<td>323</td>
<td>219</td>
<td>211</td>
</tr>
<tr>
<td>8</td>
<td>196</td>
<td>62</td>
<td>217</td>
<td>82</td>
<td>188</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression
Table 21: Comparison of execution times for indexing schemes for cyclic distributed template using virtual cyclic view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>145</td>
<td>145</td>
<td>150</td>
<td>59</td>
<td>185</td>
</tr>
<tr>
<td>2</td>
<td>328</td>
<td>328</td>
<td>296</td>
<td>144</td>
<td>364</td>
</tr>
<tr>
<td>3</td>
<td>296</td>
<td>296</td>
<td>185</td>
<td>150</td>
<td>184</td>
</tr>
<tr>
<td>4</td>
<td>651</td>
<td>651</td>
<td>587</td>
<td>328</td>
<td>724</td>
</tr>
<tr>
<td>5</td>
<td>366</td>
<td>365</td>
<td>184</td>
<td>184</td>
<td>182</td>
</tr>
<tr>
<td>6</td>
<td>587</td>
<td>587</td>
<td>365</td>
<td>296</td>
<td>365</td>
</tr>
<tr>
<td>8</td>
<td>1298</td>
<td>1297</td>
<td>1170</td>
<td>650</td>
<td>1444</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

that without hole compression. The smaller the stride, the greater the improvement. This follows by noting that the effective access stride for the allocation scheme with hole compression is $s$ while that for the allocation scheme without hole compression is $c \times s$. When the effective stride is greater than four both schemes have nearly equal performance. For the virtual cyclic view, under the allocation scheme with hole compression, all the elements in a virtual processor are stored consecutively in main memory. Under the allocation scheme without hole compression the stride between consecutive elements mapped to a virtual processor is $b \times c$. For $b = 720$ and $b = 80$, (Table 19 and Table 20) and for all the data points observed, the scheme with hole compression performs better than the scheme without hole compression. This benefit arises primarily due to the increase in spatial locality. For the cyclic distribution (Table 21), when the alignment stride $s$ and the number of processors $P$ are not relatively prime, the template cells are mapped such that either all the template cells
mapped to a processor have array elements aligned with them or all the template cells are empty. Thus the local memory storage with and without hole compression is identical. Hence for \( c = 2 \) and \( c = 8 \), the execution times for the two schemes are nearly identical. For all the other points the scheme with hole compression performs better than the scheme without hole compression.

### Table 22: Comparison of memory wasted for block distributed template under the virtual block view. \( P = 32, M = 23040 \).

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>720</td>
<td>0</td>
<td>1440</td>
<td>0</td>
<td>2880</td>
</tr>
<tr>
<td>2</td>
<td>1440</td>
<td>0</td>
<td>2880</td>
<td>0</td>
<td>5760</td>
</tr>
<tr>
<td>3</td>
<td>2160</td>
<td>0</td>
<td>4320</td>
<td>0</td>
<td>8640</td>
</tr>
<tr>
<td>4</td>
<td>2880</td>
<td>0</td>
<td>5760</td>
<td>0</td>
<td>11520</td>
</tr>
<tr>
<td>5</td>
<td>3600</td>
<td>0</td>
<td>7200</td>
<td>0</td>
<td>14400</td>
</tr>
<tr>
<td>6</td>
<td>4320</td>
<td>0</td>
<td>8640</td>
<td>0</td>
<td>17280</td>
</tr>
<tr>
<td>8</td>
<td>5760</td>
<td>0</td>
<td>11520</td>
<td>0</td>
<td>23040</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

We now present data to measure the amount of memory wasted due to the two allocation schemes. Table 22 and Table 23 present the maximum number of holes in the local memory array for a block distributed template under the virtual block view and a template distributed using a \( cyclic(80) \) distribution under the virtual block view, respectively. The data for all the other cases (\( cyclic \) under virtual cyclic view, \( cyclic(720) \) under virtual cyclic and virtual block views, and \( cyclic(80) \) under virtual cyclic view) is identical to that in Table 22. For all the cases except for the case
Table 23: Comparison of memory wasted for indexing schemes for $cyclic(80)$ distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
<td>Com</td>
<td>Non</td>
</tr>
<tr>
<td>1</td>
<td>720</td>
<td>0</td>
<td>1440</td>
<td>9</td>
<td>2880</td>
</tr>
<tr>
<td>2</td>
<td>1440</td>
<td>0</td>
<td>2880</td>
<td>18</td>
<td>5760</td>
</tr>
<tr>
<td>3</td>
<td>2160</td>
<td>0</td>
<td>4320</td>
<td>27</td>
<td>8640</td>
</tr>
<tr>
<td>4</td>
<td>2880</td>
<td>0</td>
<td>5760</td>
<td>36</td>
<td>11520</td>
</tr>
<tr>
<td>5</td>
<td>3600</td>
<td>0</td>
<td>7200</td>
<td>45</td>
<td>14400</td>
</tr>
<tr>
<td>6</td>
<td>4320</td>
<td>0</td>
<td>8640</td>
<td>54</td>
<td>17280</td>
</tr>
<tr>
<td>8</td>
<td>5760</td>
<td>0</td>
<td>11520</td>
<td>72</td>
<td>23040</td>
</tr>
</tbody>
</table>

Non - without hole compression, Com - with hole compression

in which the template is distributed using a $cyclic(80)$ distribution, under the local memory allocation scheme with hole compression, no memory is wasted. The amount of memory wasted under the allocation scheme with no hole compression increases with alignment stride $c$ and section stride $s$. This follows by noting that the template has $M \ast c \ast s$ elements and the array has $M \ast s$ elements. Assuming that the number of holes are equally distributed on the processors, the number of holes per processor is $\lceil \frac{M\ast s - M\ast s}{P} \rceil = \lceil \frac{M\ast s(e-1)}{P} \rceil$. Thus the percentage of memory wasted is $\frac{e-1}{c} \ast 100\%$.

We now present the memory allocation time for the indexing scheme with hole compression. This represents the time for performing the additional computation for identifying the section of array elements mapped to a virtual processor and evaluating the distribution parameters for each virtual processor. This computation is performed at compile-time for statically allocated arrays and at run-time for dynamically allocatable arrays. Since the memory allocation time for the allocation scheme without
Table 24: Memory allocation time for allocation schemes with hole-compression for *block* distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>25</td>
<td>24</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>24</td>
<td>28</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
<td>25</td>
<td>24</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>24</td>
<td>30</td>
<td>26</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 25: Memory allocation time for allocation schemes with hole-compression for *cyclic(720)* distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>35</td>
<td>46</td>
<td>52</td>
<td>63</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>52</td>
<td>75</td>
<td>89</td>
<td>136</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>69</td>
<td>104</td>
<td>120</td>
<td>155</td>
</tr>
<tr>
<td>4</td>
<td>51</td>
<td>88</td>
<td>113</td>
<td>155</td>
<td>283</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>105</td>
<td>163</td>
<td>270</td>
<td>250</td>
</tr>
<tr>
<td>6</td>
<td>66</td>
<td>121</td>
<td>189</td>
<td>309</td>
<td>379</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>238</td>
<td>336</td>
<td>381</td>
<td>470</td>
</tr>
</tbody>
</table>

Table 26: Memory allocation time for allocation schemes with hole-compression for *cyclic(80)* distributed template using virtual block view. $P = 32, M = 23040$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$c = 2$</th>
<th>$c = 3$</th>
<th>$c = 5$</th>
<th>$c = 6$</th>
<th>$c = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>87</td>
<td>256</td>
<td>356</td>
<td>331</td>
<td>511</td>
</tr>
<tr>
<td>2</td>
<td>156</td>
<td>413</td>
<td>614</td>
<td>731</td>
<td>927</td>
</tr>
<tr>
<td>3</td>
<td>312</td>
<td>571</td>
<td>882</td>
<td>1047</td>
<td>1340</td>
</tr>
<tr>
<td>4</td>
<td>379</td>
<td>731</td>
<td>1134</td>
<td>1367</td>
<td>1754</td>
</tr>
<tr>
<td>5</td>
<td>445</td>
<td>895</td>
<td>1397</td>
<td>1681</td>
<td>2165</td>
</tr>
<tr>
<td>6</td>
<td>519</td>
<td>1052</td>
<td>1652</td>
<td>1996</td>
<td>2579</td>
</tr>
<tr>
<td>8</td>
<td>666</td>
<td>1370</td>
<td>2166</td>
<td>2628</td>
<td>3401</td>
</tr>
</tbody>
</table>
Table 27: Memory allocation time for allocation scheme with hole-compression for cyclic(720) distributed template using virtual cyclic view. \( P = 32, M = 23040. \)

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7807</td>
<td>6859</td>
<td>4172</td>
<td>3491</td>
<td>2650</td>
</tr>
<tr>
<td>2</td>
<td>7807</td>
<td>6842</td>
<td>4170</td>
<td>3490</td>
<td>2640</td>
</tr>
<tr>
<td>3</td>
<td>7809</td>
<td>6857</td>
<td>4160</td>
<td>3488</td>
<td>2649</td>
</tr>
<tr>
<td>4</td>
<td>7811</td>
<td>6852</td>
<td>4154</td>
<td>3498</td>
<td>2649</td>
</tr>
<tr>
<td>5</td>
<td>7785</td>
<td>6868</td>
<td>4153</td>
<td>3488</td>
<td>2648</td>
</tr>
<tr>
<td>6</td>
<td>7923</td>
<td>6851</td>
<td>4168</td>
<td>3493</td>
<td>2656</td>
</tr>
<tr>
<td>8</td>
<td>7817</td>
<td>6875</td>
<td>4167</td>
<td>3497</td>
<td>2648</td>
</tr>
</tbody>
</table>

Table 28: Memory allocation time for allocation scheme with hole-compression for cyclic(80) distributed template using virtual cyclic view. \( P = 32, M = 23040. \)

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>970</td>
<td>2875</td>
<td>566</td>
<td>1841</td>
<td>397</td>
</tr>
<tr>
<td>2</td>
<td>973</td>
<td>2872</td>
<td>563</td>
<td>1842</td>
<td>396</td>
</tr>
<tr>
<td>3</td>
<td>1008</td>
<td>2877</td>
<td>481</td>
<td>1843</td>
<td>404</td>
</tr>
<tr>
<td>4</td>
<td>970</td>
<td>2870</td>
<td>565</td>
<td>1847</td>
<td>314</td>
</tr>
<tr>
<td>5</td>
<td>971</td>
<td>2875</td>
<td>565</td>
<td>1841</td>
<td>315</td>
</tr>
<tr>
<td>6</td>
<td>972</td>
<td>2877</td>
<td>564</td>
<td>1839</td>
<td>310</td>
</tr>
<tr>
<td>8</td>
<td>975</td>
<td>2880</td>
<td>576</td>
<td>1839</td>
<td>399</td>
</tr>
</tbody>
</table>

Table 29: Memory allocation time for allocation scheme with hole-compression for cyclic distributed template using virtual cyclic view. \( P = 32, M = 23040. \)

<table>
<thead>
<tr>
<th>s</th>
<th>( c = 2 )</th>
<th>( c = 3 )</th>
<th>( c = 5 )</th>
<th>( c = 6 )</th>
<th>( c = 8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>66</td>
<td>76</td>
<td>81</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>77</td>
<td>75</td>
<td>71</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
<td>76</td>
<td>76</td>
<td>78</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>76</td>
<td>73</td>
<td>68</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>79</td>
<td>73</td>
<td>70</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>56</td>
<td>78</td>
<td>75</td>
<td>72</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>79</td>
<td>77</td>
<td>75</td>
<td>63</td>
</tr>
</tbody>
</table>
hole-compression is negligible, we present only the allocation times for the allocation scheme with hole compression. Tables 24 to 29, present the memory allocation time for different block sizes and virtual views. It can be observed that for the block distribution under the virtual block view, the memory allocation time is nearly independent of the strides $s$ and $c$. This follows by noting that each processor has a single virtual processor mapped to it and hence the amount of additional computation is nearly fixed. For the block-cyclic distribution under a virtual block-view, the allocation time increases with increasing stride $s$ and $c$. This increase occurs primarily due to an increase in the number of virtual processors $\left( \frac{M_{max}}{b} \right)$ which leads to an increase in the total amount of computation to be performed. For the block-cyclic distribution under the virtual cyclic view, the allocation time is nearly independent of the array section stride $s$. This follows by noting that the stride between virtual processors which have array elements mapped to them is $g = \gcd(c, P \cdot b)$ (Section 4.3.2).

Thus the approximate number of such virtual processors per physical processor is $\frac{P \cdot b}{g}$, which is independent of the stride $s$. Since the memory allocation computation is performed only for these processors, the allocation time is independent of $s$. When $b = 720$, we have $\gcd(P \cdot b, c) = c$. Thus the memory allocation time decreases with an increase in $c$. For $b = 80$, the allocation time varies inversely to the value of the $\gcd$. 
On the basis of the presented data we can make the following conclusions about the allocation and indexing schemes with and without hole compression.

- The allocation scheme with hole compression requires greater allocation time than the scheme without hole compression. However, for most the cases considered, the hole-compression scheme eliminates nearly all holes and leads to significant savings in memory usage.

- The table generation time for the execution of the array statement for the indexing scheme with hole-compression is greater than that for the scheme without hole-compression. However, the scheme with hole compression often has better execution times than that without hole compression primarily due to better spatial locality.

The allocation scheme of choice would depend on the relative importance of memory wastage reduction as compared to execution time reduction. For instance, if sufficient amount of memory is available, then no hole-compression need be performed leading to lower compile and execution times. However, if memory is in shortage, then it is important to compress holes and pay the additional cost of a higher execution time and compile-time for the array statement.

4.5 Summary

In this chapter, we have addressed the memory allocation and index set enumeration problem for arrays mapped using a two-level mapping with non-unit stride to a set of processors. Using simple mathematical properties of regular sections we identified
the regular section of array elements mapped to a processor for block and cyclically
distributed arrays. This regular section characterization identifies the exact amount
of the memory required on the processor for block and cyclically distributed arrays.
Furthermore it facilitates the index set translation from the local template space to
the local compressed array space. We have identified methods for extending these
techniques to the virtual processor approach for handling block-cyclically distributed
arrays. Extensive experimental evaluation on the Cray T3D demonstrates that while
the scheme with hole allocation will require additional computation at compile time, it
leads to significant savings in memory used. Furthermore, while the table generation
times for the execution of the array statement, would increase, the time for execution
of the array statement is reduced due to an improvement in spatial locality.
Array redistribution is used in languages such as High Performance Fortran (HPF) [46, 86, 84, 90], Fortran D [47], and Vienna Fortran [26, 28, 29] to dynamically change the distribution of an array from a specified source to a specified target distribution. Redistribution of data arrays is used under the following circumstances.

- Different parts of a program vary in their access patterns to a shared array. Thus a different data distribution of the same array is best suited for different parts of the program. For instance, the alternate direction implicit (ADI) method [64] for solving two-dimensional diffusion equations consists of two phases - the first phase accesses a two-dimensional array along the rows and the second phase along the columns. A distribution in which the array rows are local to a processor and the columns are distributed, eliminates communication for the first phase and allows the processors to operate in parallel. Similarly, a distribution in which the array columns are local and the rows are distributed, eliminates communication for the second phase. Thus the need to explicitly redistribute the array between the two phases arises. Distributed-memory implementations
of several scientific applications including the multi-dimensional fast Fourier transform [127] use array redistribution.

- Scientific libraries provide peak performance for a fixed distribution of arrays included in the formal parameters. These distributions may not conform with the distributions of the actual parameters passed at library routine invocation, leading to performance degradation. Thus the actual array parameters should be explicitly redistributed across the program-library interface. Similarly array redistribution is also required when the distributed arrays passed as arguments to runtime libraries or subroutines have distributions different from those of the formal parameters.

- Data redistribution is implicitly required in the execution of array statements, where the array sections in the statement are the entire arrays and have different distributions.

After changing the distribution of an array, elements located on a processor under the source distribution may be located on a different processor under the target distribution. Performing array redistribution incurs an indexing overhead for identifying the processors involved in the communication and the array elements to be communicated. Specifically each processor $p$ has to determine the following processor and data index sets:
• **Send processor set of** $p$: set of processors to which $p$ has to send data.

• **Send data index set of** $p$ to processor $q$: indices of the array elements which are resident on $p$ under the source distribution but are resident on $q$ after redistribution.

• **Receive processor set of** $p$: set of processors from which $p$ has to receive data.

• **Receive data index set of** $p$ from processor $q$: indices of the array elements which are resident on $q$ under the source distribution but are resident on $p$ after redistribution.

A closed form characterization of these sets would reduce the indexing overhead for packing data into messages on the sending processor and unpacking data at the receiving processor. Such a characterization in terms of simple regular sections is possible for array redistributions restricted to block and cyclically distributed arrays [58, 82]. However, in general for the block-cyclic distribution, such a closed form characterization is not possible.

In this chapter, we develop a precise closed form characterization of the processor and data index sets for the special cases of array redistribution when either the source block size is a multiple of the target block size or vice versa. This closed form characterization facilitates the development of a communication and indexing cost model for array redistribution and a distributed scheduling algorithm which eliminates node contention for the communication arising due to array redistribution.
Extensions of the developed closed forms to handle redistribution of multi-dimensional arrays are provided.

This chapter is organized as follows. Section 5.1 briefly describes array redistribution. Section 5.2 describes the tensor product representation which is used as a mixed radix representation for the data distributions and index sets for array redistribution. The mixed radix representation is provided in Section 5.3. In Section 5.4, closed forms for the send and receive processor and data index sets for cases when the source block size is a multiple of the target block size and vice versa are presented. Extensions for redistribution of multi-dimensional arrays are described.

5.1 Array Redistribution

In this section we briefly describe the array redistribution directive in HPF and different methods for executing it on distributed-memory machines. We focus on identity alignments of arrays with templates. For non-identity alignments, array redistribution is equivalent to the execution of the array statement and techniques developed in Chapter III and Chapter IV can be used in this case. For the identity alignment, the relationships between the global index, the local index and the processor index for regular data distributions of a one-dimensional array are shown in Table 30. The indexing for arrays $A$ and $A_{loc}$ begins at zero and the processors are number from $0$ to $P - 1$.

Array redistribution implies a change in the distribution of the array. For instance, consider an array $B(0 : 14)$ distributed on three processors using a block distribution.
Table 30: Index mapping functions for regular data distributions.

<table>
<thead>
<tr>
<th></th>
<th>block</th>
<th>cyclic</th>
<th>cyclic(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local to global</td>
<td>$k = p([N/P]) + l$</td>
<td>$k = lP + p$</td>
<td>$k = (l \div b)bP + bp + l \mod b$</td>
</tr>
<tr>
<td>global to local</td>
<td>$l = k \mod [N/P]$</td>
<td>$l = k \div P$</td>
<td>$l = (k \div Pb)b + k \mod b$</td>
</tr>
<tr>
<td>global to proc</td>
<td>$p = k \div [N/P]$</td>
<td>$p = k \mod P$</td>
<td>$p = (k \div b) \mod P$</td>
</tr>
</tbody>
</table>

$k$ global index $0 \leq k \leq N - 1$; $l$ local index $0 \leq l < [b \ast ([N/(Pb)])$; $p$ processor $0 \leq p < P$.

REAL A(100,100)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE (BLOCK,CYCLIC(2)) ONTO P:: A
!HPF$ DYNAMIC A
...
    FORALL (i=2:98, j=2:98)
        A(i,j) = A(i-1,j-1) + A(i+1,j+1)
...
!HPF$ REDISTRIBUTE A(CYCLIC(2),BLOCK)

Figure 39: HPF program segment.

The redistribution of $B$ to a cyclic distribution is illustrated in Fig. 40. Fig. 39, illustrates the syntax of the REDISTRIBUTE command. After redistribution, array elements located on a processor under the source distribution may be located on a different processor under the target distribution. Thus a processor has to determine the set of array elements to send to and receive from other processors. Using the communication index sets defined in Chapter II, the node program pseudo-code for array redistribution is as shown in Fig. 41. Note that no particular order of the
message sends and receives is implied in Fig. 41.

For array redistributions restricted to block and cyclic distributions, the processor and data index sets can be expressed as simple regular sections [58, 82]. However, in general for block-cyclic distributions the index sets cannot be expressed as regular sections. In this chapter, we develop closed form expressions for special cases of array redistribution involving block-cyclic arrays. The general case of array redistribution can be expressed in terms of the special cases.

5.2 The Tensor Product Representation

We use the tensor product representation as a mixed radix representation for regular data distributions to develop and represent the index sets for redistribution. We provide a brief overview of the tensor product representation for regular data distributions. The reader is referred to [56] for details.

Let $A$ be an $m \times n$ matrix and $B$ a $p \times q$ matrix. The tensor product of $A$ and $B$ is the block matrix obtained by replacing each element $a_{i,j}$ of matrix $A$ by $a_{i,j}B$. 

<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>0 1 2 3 4</td>
<td>5 6 7 8 9</td>
<td>10 11 12 13 14</td>
</tr>
<tr>
<td>Cyclic</td>
<td>0 3 6 9 12</td>
<td>1 4 7 10 13</td>
<td>2 5 8 11 14</td>
</tr>
</tbody>
</table>

Figure 40: Redistribution of array $A(0:14)$ from block to cyclic distribution.
/* Node code for processor p */

/* Sending phase */
for \( q \in PSend(p) \)
    
    \( Tmp(0 : |DSend(p, q)| - 1) = A.loc(DSend(p, q)) \)
    
endfor

/* Receiving phase */
for \( q \in PRecv(p) \)
    
    \( recv(q, Tmp, |DRecv(p, q)|) \)
    
    \( A.loc(DRecv(p, q)) = Tmp(0 : |DRecv(p, q)| - 1) \)
endfor

Figure 41: Array redistribution using index sets.

\[ A \otimes B = \begin{bmatrix} a_{0,0}B & \cdots & a_{0,n-1}B \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B & \cdots & a_{m-1,n-1}B \end{bmatrix}_{mp \times nq} \]  

(5.1)

We use the tensor product of vector bases to represent regular data distributions.

A vector basis of size \( m \) consists of basis vectors \( e_k^m \), \( 0 \leq k < m \), where \( e_k^m \) is a column vector of length \( m \) with one at position \( k \) and zeroes elsewhere. We use \( e_i^m \) to represent the vector basis of size \( m \). The tensor product \( e_i^m \otimes e_j^n \) of two vector bases, \( e_i^m \) and \( e_j^n \) is called a tensor basis and from the definition of the tensor product we have: \( e_i^m \otimes e_j^n = e_{in+j}^{mn} \). Thus the size of the tensor basis \( e_i^m \otimes e_j^n \) is \( mn \). Expressing a vector basis \( e_i^M \) as the tensor product of vector bases \( e_i^{m_{i-1}} \otimes \cdots \otimes e_i^{m_0} \) is called the factorization of the vector basis, where \( M = m_0 \times \cdots \times m_{l-1} \) and \( i_k = (i \div m_{k-1}) \mod m_k, M_k = \Pi_{i=0}^{k} m_i, M_{-1} = 1 \). For example, some of the pos-
sible factorizations of the vector basis $e_{i_2}^{12}$ are the tensor basis $e_{i_2}^{2} \otimes e_{i_1}^{3} \otimes e_{i_0}^{2}$ where $i_2 = (i \text{ div } 6)$, $i_1 = (i \text{ div } 2) \mod 3$, and $i_0 = i \mod 2$ or the tensor basis $e_{i_1}^{4} \otimes e_{i_0}^{3}$ where $i_1 = i \text{ div } 3$ and $i_0 = i \mod 3$. Factorizing a vector basis in a tensor basis is referred to as factorization of the tensor basis. Expressing a tensor basis $e_{i_{l-1}}^{m_{l-1}} \otimes \cdots \otimes e_{i_0}^{m_0}$ as a vector basis $e_i^M$, $i = \sum_{k=0}^{l-1} i_k M_{k-1}$ is called linearization of the tensor basis. For example, the tensor basis $e_{i_1}^{4} \otimes e_{i_0}^{3}$ can be linearized to give the vector basis $e_{3i_1+i_0}^{12}$.

The tensor product basis provides a natural representation for indexing into segmented or multi-dimensional arrays. We now define the semantics of regular data distributions using the tensor basis.

5.3 A Mixed Radix Representation of Regular Data Distributions

5.3.1 Representation of One-Dimensional Arrays

Consider the array $A(0 : N - 1)$ distributed over $P$ processors numbered from 0 to $P - 1$. We assume that $N$ is a multiple of $P^1$. A mixed radix representation of a regular data distribution of the array $A$ is obtained by associating index $k$ of $A$ with the vector basis element $e_k^N$ and factorizing the vector basis. Associating array indices with the indices of the vector basis helps in identifying the processor index and the local index when the array is partitioned due to the data distribution. For example, under a block distribution the array is partitioned into segments of size $\frac{N}{P}$. The relationship between the global index $k$, the processor index $p$, and the local

\[ ^1\text{If } P \text{ does not evenly divide } N, \text{ then the array is assumed to be padded to } \lceil N/P \rceil P \text{ elements. No memory is actually allocated for the dummy elements.} \]
index $l$ as shown in Table 30 can be represented by the equality:

$$e_k^N = e_p^P \otimes e_l^N,$$

where $p = k \div \frac{N}{P}$ and $l = k \mod \frac{N}{P}$. In the above equation, the index of vector basis $e_p^P$ is associated with the processor index on which element $A(k)$ is located after being distributed using a block distribution. We distinguish the vector basis corresponding to the processor index by denoting it as $\rho$ and refer to it as a processor basis. The index of the vector basis $e_i^{N/P}$ is associated with the local index of $A(k)$ in $A.loc$. The resulting tensor basis $\rho_p^P \otimes e_i^{N/P}$ provides a mixed radix representation for the block distribution. A tensor basis in which the vector basis associated with the processor index is denoted as $\rho$ is referred to as a distribution basis. Next, consider the cyclic(B) distribution of array $A$. The array is divided into $N/B$ blocks of size $B$ each, which are cyclically assigned to $P$ processors. Let $C = \frac{N}{PB}$ be the number of courses (number of blocks) on each processor. The relation between the global index $k$, the processor index $p$, and the index $b$ within a block is represented by:

$$e_k^N = e_c^C \otimes \rho_p^P \otimes e_b^B$$

where $c = k \div PB$, $p = (k \div B) \mod P$, and $b = k \mod B$. In the distribution basis, $c$ is the course index and $p$ is the processor index. Note that $b$ does not correspond to the local index $l$. The local index $l$ is determined by a combination of the course index $c$ and the block index $b$ as shown in Def. 5.1: $l = Bc + b$. We define the indexing functions for the distribution basis which select the vector bases of the distribution which identify the processor index $p$, the local index $l$ and the global
index $k$ of each index of the distributed array. We consider the distribution basis for the block-cyclic distribution as it is the most general regular data distribution.

**Definition 5.1 (Indexing functions)** For a one-dimensional array $A(0 : N - 1)$ let the distribution basis be $e^C_c \otimes \rho^P_p \otimes e^B_b$, then the global index function $\text{global}$, the processor index function $\text{proc}$, and the local index function $\text{local}$ are

$$\text{global}(e^C_c \otimes \rho^P_p \otimes e^B_b) = e^N_{PBc+Bp+b} \quad (5.4)$$

$$\text{proc}(e^C_c \otimes \rho^P_p \otimes e^B_b) = \rho^P_p \quad (5.5)$$

$$\text{local}(e^C_c \otimes \rho^P_p \otimes e^B_b) = e^B_{Bc+b} \quad (5.6)$$

Note that since $c = k \text{ div } PB$ and $b = k \text{ mod } b$, Def. 5.1 implies the relations between the indices illustrated in Table 30, i.e., $k = PBc + Bp + b$ and $l = Bc + b$.

### 5.3.2 Representation of Multi-Dimensional Arrays

The distribution basis for a multi-dimensional array can be expressed as a tensor product of the distribution bases for each dimension. Consider an array $A(0 : N_2 - 1, 0 : N_1 - 1)$ stored using a row major order, i.e., element $A(i, j)$ has an index $iN_1 + j$ in linear memory. Since $e^N_{i_2} \otimes e^N_{i_1} = e^N_{i_2i_1+j}$, the index $(i, j)$ for the two-dimensional array can be associated with the tensor basis index $e^N_{i_2} \otimes e^N_{j_1}$. Similarly the distribution of each dimension can be independently specified by expressing the vector bases $e^N_{i_2}$ and $e^N_{j_1}$ as distribution bases. Thus the distribution basis for a multi-dimensional array can be represented as the tensor product of the distribution bases for each dimension. For instance, the $(\text{cyclic}(B_2), \text{cyclic}(B_1))$ distribution of
the \( N_2 \times N_1 \) array \( A \) onto a \( P_2 \times P_1 \) processor mesh is represented by the distribution basis

\[
[e_{c_2}^{C_2} \otimes \rho_{p_2}^{P_2} \otimes e_{b_2}^{B_2}] \otimes [e_{c_1}^{C_1} \otimes \rho_{p_1}^{P_1} \otimes e_{b_1}^{B_1}] \tag{5.7}
\]

where \( C_2 = \frac{N_2}{P_2 B_2} \) and \( C_1 = \frac{N_1}{P_1 B_1} \) are the number of courses for the block-cyclic distribution along the first and second dimension, respectively. The square parenthesis separate the distribution bases corresponding to each dimension. The indexing functions for the distribution basis of a multi-dimensional array can be defined in terms of those for one-dimensional arrays.

**Definition 5.2 (Indexing functions)** For a multi-dimensional array \( A(0 : N_{t-1} - 1, \ldots, 0 : N_0 - 1) \), with a distribution basis \( \delta_i \) for the distribution along dimension \( i \), the global index function \( \text{global} \), the processor index function \( \text{proc} \) and the local index function \( \text{local} \) are defined as follows:

\[
\text{global}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) = [\text{global}(\delta_{t-1})] \otimes \cdots \otimes [\text{global}(\delta_0)] \tag{5.8}
\]

\[
\text{proc}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) = [\text{proc}(\delta_{t-1})] \otimes \cdots \otimes [\text{proc}(\delta_0)] \tag{5.9}
\]

\[
\text{local}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) = [\text{local}(\delta_{t-1})] \otimes \cdots \otimes [\text{local}(\delta_0)] \tag{5.10}
\]

**5.3.3 Array Redistribution**

Array redistribution corresponds to a remapping of the processor basis in the distribution basis. If \( \delta_s \) is the initial distribution basis and \( \delta_t \) is the distribution basis after the redistribution then \( \text{global}(\delta_s) = \text{global}(\delta_t) \), i.e., the global index is unchanged. However, \( \text{proc}(\delta_s) \) and \( \text{proc}(\delta_t) \) may be different, which implies that redistribution
involves communication. Also, local(δs) and local(δt) may be different. We refer to δs as the source basis, and to δt as the target basis. For example, δs = ρ_p^A ⊗ e^A_i represents a block distribution of A(0 : 15) on four processors. Remapping the processor basis gives the new distribution basis δt = e^A_p ⊗ ρ^A_l which corresponds to a cyclic distribution on four processors. Clearly global(δs) = global(δt) = e_{4p+1}^{16}. But, proc(δs) = ρ^A_p and proc(δt) = ρ^A_l, which implies that communication may be required since p and l can be different for a global array index k. Also, local(δs) = e^A_i and local(δt) = e^A_p which implies that the local index of an element can change after redistribution.

5.3.4 Index Set Representation

We represent the processor and data index sets for array redistribution in terms of simple regular sections or a tensor basis augmented with ranges for the indices of the vector bases. A regular section (a : b : s) represents the indices \{i | i = a + js, 0 ≤ j < \left\lfloor \frac{b-a}{s} \right\rfloor\}. A tensor product basis of the form e^C_s ⊗ e^b_p represents all the indices \{i | i = i_1b + p, 0 ≤ i_1 < C\}. The symbol * is used to represent all the index values from 0 to C - 1 where C is the size of the corresponding vector basis. In the above example, the index i_1 iterates over all the values represented by the * symbol. The set of indices represented by a tensor product basis can be enumerated by a perfect loop nest. A for loop is generated for every vector basis in the tensor basis with either a * as index or an index for which a range is specified. If a * is specified, the lower loop bound is 0 and the upper loop bound is equal to one less than the size of the corresponding vector basis. If a range is specified then the loop scans the specified range. For example, the indices i specified by e^C_s ⊗ e^Y_y ⊗ e^b_s, y ∈ (0 : Y - 1 : P) can
for $i_1 = 0, C - 1$ /* loop for $e^C_*$ */
for $y = 0, Y - 1, P$ /* loop for $e^Y_y$, $y \in (0 : Y - 1 : P)$ */
for $i_2 = 0, b - 1$ /* loop for $e^b_*$ */
  $i = i_1 * Y * b + y * b + i_2$
endfor
endfor
endfor

Figure 42: Loop nest for enumerating index set $e^C_* \otimes e^Y_y \otimes e^b_*$, $y \in (0 : Y - 1 : P)$.

be enumerated by the loop nest in Fig. 42.

We now develop the processor and data index sets for two special cases of array redistribution involving block-cyclically distributed arrays.

5.4 Index Sets for Array Redistribution

In this section, we develop the processor and data index sets for the special cases when the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. We first consider one-dimensional arrays and then describe the extensions for multi-dimensional arrays.

5.4.1 Array Redistribution of One-Dimensional Arrays

In the following sections we consider an array $A(0 : N - 1)$ distributed onto $P$ processors.
Array Redistribution from a Cyclic(Yt) to a Cyclic(t) distribution

Let the source distribution be cyclic(Yt) and the target distribution be cyclic(t). Let \( C = \frac{N}{PYt} \) be the number of courses in the source block-cyclic distribution. The send processor and data index sets are as follows:

**Theorem 5.1** For an array redistribution from a cyclic(Yt) distribution to a cyclic(t) distribution, the send processor set for processor \( p \) is

\[
PSend(p) = \begin{cases} 
((pY) \mod P : (pY + Y - 1) \mod P) & \text{if } Y < P \\
(0 : P - 1) & \text{if } Y \geq P. 
\end{cases}
\] (5.11)

For a processor \( q \in PSend(p) \), the send data set from processor \( p \) to processor \( q \) is

\[
DSend(p, q) = \begin{cases} 
e^C \otimes e^Y_{y_1} \otimes e^t_1 & \text{where } y_1 = (q - pY) \mod P \\
e^C \otimes e^Y_{y_1} \otimes e^t_1 & \text{where } y_1 = ((q - pY) \mod P : Y - 1 : P) 
\end{cases}
\] (5.12)

**Proof:** The source distribution is represented by the distribution basis \( \delta_s = e^C_{c_1} \otimes \rho^P_{p_1} \otimes e^Y_{y_1} \otimes e^t_1 \) and the target distribution basis is \( \delta_t = e^C_{c_2} \otimes e^Y_{y_2} \otimes \rho^P_{p_2} \otimes e^t_2 \). Note that the trailing vector basis has to be further factorized to facilitate the following analysis. Redistribution requires that \( \text{global}(\delta_s) = \text{global}(\delta_t) \) which gives the equality

\[
c_1 * PYt + p_1 * Yt + y_1 * t + t_1 = c_2 * PYt + y_2 * Pt + p_2 * t + t_2 \] (5.13)

\[
c_1 * PYt + (p_1 * Y + y_1) * t + t_1 = c_2 * PYt + (y_2 * P + p_2) * t + t_2 \] (5.14)

Since \( 0 \leq t_1, t_2 < t \) and \( 0 \leq c_1, c_2 < C \), we have \( c_1 = c_2, t_1 = t_2 \) which implies that \( p_1 * Y + y_1 = y_2 * P + p_2. \)
To determine the send processor set for a processor \( p_1 \), we need to determine all the unique values for \( p_2 = (y_1 + p_1 \cdot Y) - y_2 \cdot P \) where \( 0 \leq y_1, y_2 < Y \). Consider the case when \( Y < P \). Since \( 0 \leq p_2 < P \) and \( Y < P \), we have \( p_2 = (y_1 + p_1 \cdot Y) - y_2 \cdot P = (y_1 + p_1 \cdot Y) \mod P, 0 \leq y_1 < Y \). Thus \( P_{Send}(p_1) = ((y_1 \mod Y) \mod P : (p_1 \cdot Y + Y - 1) \mod P) \), if \( Y < P \). Note that \( P_{Send}(p_1) \) may include a wrap-around depending on whether \( (p_1 \cdot Y) \mod P > (p_1 \cdot Y + Y - 1) \mod P \), i.e., the set of processors in \( P_{Send}(p_1) \) will be \( ((y_1 \mod Y) \mod P : P - 1) \cup (0 : (p_1 \cdot Y + Y - 1) \mod P) \).

Consider the case when \( Y \geq P \). Let \( x = p_1 \cdot Y \mod P \). Since \( p_1 \cdot Y < P \cdot Y \land 0 \leq y_2 < Y \Rightarrow \exists y_2' \), such that \( p_1 \cdot Y - y_2' \cdot P = x \). Fig. 43 illustrates the relation among the values. Since \( p_2 = y_1 + (p_1 \cdot Y - y_2 \cdot P) \), \( p_2 \) can take the values from \( x \) to \( P - 1 \) for \( y_2 = y_2' \) and \( y_1 \in (0 : P - x - 1) \). Similarly, \( p_2 \) can take the values from \( 0 \) to \( x - 1 \) for \( y_2 = y_2' - 1 \) and \( y_1 \in (P - x : P - 1) \). Thus when \( Y \geq P \), \( P_{Send}(p_1) = (0 : P - 1) \).

We now determine the send data index set from a processor \( p_1 \) to a processor \( p_2 \). We obtain all the unique values for \( y_1 \) such that \( y_1 = y_2 \cdot P + (p_2 - p_1 \cdot Y) \). The smallest non-negative value for \( y_1 \) is given by \( y_1 = (p_2 - p_1 \cdot Y) \mod P \). If \( Y < P \), then there is just a single value for \( y_1 \). If \( Y \geq P \), then \( y_1 \in ((p_2 - p_1 \cdot Y) \mod P : Y - 1 : P) \). Since the data send index set consists of local indices on processor \( p_1 \),
\[ DSend(p_1, p_2) = local(e_C^* \otimes e_p^R \otimes e_2^Y \otimes e_3^t) = e_C^* \otimes e_2^Y \otimes e_3^t. \] Thus we have

\[
DSend(p, q) = \begin{cases} 
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{where } y_1 = (q - pY) \mod P \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{where } y_1 = ((q - pY) \mod P : Y - 1 : P) \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{if } Y < P \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{if } Y \geq P.
\end{cases}
\] (5.15)

Note that the processor send set may involve a wrap-around at \( P - 1 \) if \( pY \mod P > (pY + Y - 1) \mod P \). The receive processor and data index sets for a cyclic(Yt) to cyclic(t) array redistribution are:

**Theorem 5.2** For an array redistribution from a cyclic(Yt) distribution to a cyclic(t) distribution, the receive processor set for a processor \( p \) is

\[
PRecv(p) = \begin{cases} 
  (p :YP - 1 : P) \div Y & \text{if } Y < P \\
  (0 : P - 1) & \text{if } Y \geq P.
\end{cases}
\] (5.16)

For a processor \( q \in PRecv(p) \), the receive data index set is

\[
DRecv(p, q) = \begin{cases} 
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{where } y_2 = \left\lfloor \frac{qY - p}{P} \right\rfloor \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{where } y_2 = \left( \left\lfloor \frac{qY - p}{P} \right\rfloor : \left\lfloor \frac{qY - p + Y - 1}{P} \right\rfloor \right) \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{if } Y < P \\
  e_C^* \otimes e_2^Y \otimes e_3^t & \text{if } Y \geq P.
\end{cases}
\] (5.17)

**Proof:** The source distribution is represented by the distribution basis \( \delta_s = e_C^* \otimes \rho_p^R \otimes e_2^Y \otimes e_3^t \) and the target distribution basis is \( \delta_t = e_C^* \otimes \rho_p^R \otimes e_2^Y \otimes e_3^t \). Note that the trailing vector basis has to be further factorized to facilitate the following analysis. Redistribution requires that \( global(\delta_s) = global(\delta_t) \) which gives the equality

\[
c_1 \ast PYt + p_1 \ast Yt + y_1 \ast t + t_1 = c_2 \ast PYt + y_2 \ast Pt + p_2 \ast t + t_2 \] (5.18)
\[
c_1 \ast PYt + (p_1 \ast Y + y_1) \ast t + t_1 = c_2 \ast PYt + (y_2 \ast P + p_2) \ast t + t_2 \] (5.19)
Since $0 \leq t_1, t_2 < t$ and $0 \leq c_1, c_2 < C$, we have $c_1 = c_2$, $t_1 = t_2$ which implies that $p_1 \star Y + y_1 = y_2 \star P + p_2$.

For the receive processor set, we need to determine the unique values for $p_1$, where $p_1 \star Y + y_1 = y_2 \star P + p_2 = ((y_2 \star P + p_2) \text{div } Y) \star Y + (y_2 \star P + p_2) \mod Y$. It follows that $p_1 = (y_2 \star P + p_2) \text{div } Y$, $0 \leq y_2 < Y$. Thus we have $p_1 \in \left( (p_2 : YP - 1 : P) \text{div } Y \right)$. When $Y \geq P$ the set $(p_2 : YP - 1 : P) \text{div } Y$ includes all processors $(0 : P - 1)$.

To determine the receive data index set for a processor $p_2$ from a processor $p_1$, we need to determine the set of all unique $y_2$ s.t., $y_2 \star P = (p_1 \star Y - p_2) + y_1$, $0 \leq y_1 < Y$. The smallest non-negative value for $y_2$ is $\left\lceil \frac{p_1 \star Y - p_2}{p} \right\rceil$. When $Y < P$, there is a single value for $y_2$. If $Y \geq P$, $y_2$ can take values in the range $\left\lceil \frac{p_1 \star Y - p_2}{p} \right\rceil : \left\lceil \frac{p_1 \star Y - p_2 + Y - 1}{p} \right\rceil$. Since, the data receive index set consists of local indices on processor $p_2$, $D\text{Recv}(p_2, p_1) = \text{local}(e_C \otimes e^P_{p_2} \otimes e^Y_{y_2} \otimes e^t_*) = e_C \otimes e^Y_{y_2} \otimes e^t_*$. Thus we have

$$D\text{Recv}(p, q) = \begin{cases} \begin{align*} e_C \otimes e^Y_{y_2} \otimes e^t_* & \text{ where } y_2 = \left\lfloor \frac{qY - p}{p} \right\rfloor \\
 & \text{ if } Y < P \\
\end{align*} \end{cases} \begin{align*} e_C \otimes e^Y_{y_2} \otimes e^t_* & \text{ where } y_2 = \left( \left\lceil \frac{qY - p}{p} \right\rceil : \left\lceil \frac{qY - p + Y - 1}{p} \right\rceil \right) \\
 & \text{ if } Y \geq P. \end{align*}$$

**Array Redistribution from a Cyclic(s) to a Cyclic(Ys) distribution**

Let the source distribution be cyclic($s$), the target distribution be cyclic($Ys$), and let $C = \frac{N}{PY_s}$ denote the number of courses in the source block-cyclic distribution. The processor and data index sets for a cyclic($s$) to cyclic($Ys$) redistribution can be determined in a manner similar to that for a cyclic($Yt$) to a cyclic($t$) redistribution. The send processor and data index sets for a redistribution from a cyclic($s$) to a
cyclic(Ys) distribution are equivalent to the receive processor and data index sets for the cyclic(Yt) to a cyclic(t) redistribution, respectively and vice versa. We provide the processor and data index sets without proofs.

**Theorem 5.3** For an array redistribution from a cyclic(s) distribution to a cyclic(Ys) distribution, the send processor set for a processor p is

\[
    P_{Send}(p) = \begin{cases} 
    (p : PY - 1 : P) \text{ div } Y & \text{if } Y < P \\
    (0 : P - 1) & \text{if } Y \geq P.
    \end{cases}
\]

(5.21)

For a processor q \(\in P_{Send}(p)\), the send data set from p to q is

\[
    D_{Send}(p,q) = \begin{cases} 
    e^{c} \otimes e^{y_{1}} \otimes e^{s} & \text{where } y_{1} = \left\lfloor \frac{qY - p}{P} \right\rfloor \\
    e^{s} \otimes e^{y_{1}} \otimes e^{s} & \text{where } y_{1} = \left( \left\lfloor \frac{qY - p}{P} \right\rfloor : \left\lfloor \frac{qY - p + Y - 1}{P} \right\rfloor \right)
    \end{cases}
\]

if \(Y < P\).

(5.22)

**Theorem 5.4** For an array redistribution from a cyclic(s) distribution to a cyclic(Ys) distribution, the receive processor set for a processor p is

\[
    P_{Recv}(p) = \begin{cases} 
    ((pY) \text{ mod } P : (pY + Y - 1) \text{ mod } P) & \text{if } Y < P \\
    (0 : P - 1) & \text{if } Y \geq P.
    \end{cases}
\]

(5.23)

For a processor q \(\in P_{Recv}(p)\), the receive data set from p to q is

\[
    D_{Recv}(p,q) = \begin{cases} 
    e^{c} \otimes e^{y_{2}} \otimes e^{s} & \text{where } y_{2} = (q - pY) \text{ mod } P \\
    e^{s} \otimes e^{y_{2}} \otimes e^{s} & \text{where } y_{2} = ((q - pY) \text{ mod } P : Y - 1 : P)
    \end{cases}
\]

(5.24)

if \(Y < P\).

if \(Y \geq P\).

We now develop the index sets for redistribution of multi-dimensional arrays.
5.4.2 Array Redistribution of Multi-dimensional Arrays

The redistribution techniques for multi-dimensional arrays differ based on whether the shape of the data and processor array change during redistribution. We have the following cases:

- **Shape Preserving Redistribution**: These routines are applicable when the shape of the data and processor array are not changed during the array redistribution, i.e., the redistribution only involves changing the distribution along the dimensions of the array. For instance, the redistribution considered in the program segment in Fig. 39 is a shape preserving redistribution. The index sets for the multi-dimensional array redistribution are developed using those developed for the one-dimensional arrays.

- **Shape Modifying Redistribution**: In general, when either the data or processor array is reshaped during redistribution, techniques developed in [58] are required to efficiently enumerate the index sets. However, for particular source and target distributions, it is possible to develop closed form expressions for the index sets [74].

**Shape Preserving Redistribution**

We develop the index sets for the shape preserving redistribution. Consider an array \( A(0 : N_{t-1} - 1, \ldots, 0 : N_0 - 1) \) distributed onto a \( P_{t-1} \times \cdots \times P_0 \) processor mesh using a regular data distribution along each dimension. The distribution basis is of the form \([\delta_{t-1}] \otimes \cdots [\delta_0]\) where \( \delta_i, 0 \leq i < t \) is the distribution basis for
dimension $i$. After redistribution, let the distribution basis be $[\delta'_i] \otimes \cdots \otimes [\delta_0]$.

Since the redistribution is shape-preserving, the shape of the data array does not change and the sizes of $\delta_i$ and $\delta'_i$ are equal. Also the sizes of $\text{proc}(\delta_i)$ and $\text{proc}(\delta'_i)$ are equal as the processor array is not reshaped. Thus the analysis of Section 5.4.1 can be independently applied for each dimension to determine the index sets for that dimension. The index sets for the multi-dimensional array redistribution are given by the tensor product of the corresponding index sets for each dimension. For instance, consider a processor $(p_{t-1}, \ldots, p_0)$, $0 \leq p_i < P_i$ and let $\text{PSend}_i(p_i)$ denote the processor send set corresponding to dimension $i$ of the array. Then the processor send set for processor $(p_{t-1}, \ldots, p_0)$ for the multi-dimensional array redistribution is $\text{PSend}_{t-1}(p_{t-1}) \otimes \cdots \otimes \text{PSend}_0(p_0)$. The index sets can be scanned using loop nests as shown in Section 5.2. Thus the node code on processor $(p_{t-1}, \ldots, p_0)$ for the multi-dimensional array redistribution is as shown in Fig. 44. For the array dimensions $i$ which are not distributed, $\text{PSend}_i$ is empty and $\text{DSend}$ includes the entire dimension.

5.5 Summary

In this chapter, we have developed a precise closed form characterization of the processor and data index sets for the special cases of array redistribution when either the source block size is a multiple of the target block size or vice versa, is developed. The general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases as is shown in Chapter VI. This closed form
/* Node code for processor p */

/* Sending phase */
for $q_{l-1} \in PSend_{l-1}(p_{l-1})$
  
  for $q_0 \in PSend_0(p_0)$
    pack $A.loc(DSend_{l-1}(q_{l-1}, q_0), ..., DSend_0(q_0, q_0))$ into $Tmp$;
    send(((q_{l-1}, ..., q_0), Tmp);
  endfor
endfor

/* Receiving phase */
for $q_{l-1} \in PRecv_{l-1}(p_{l-1})$
  
  for $q_0 \in PRecv_0(p_0)$
    recv((q_{l-1}, ..., q_0), Tmp);
    $A.loc(DRecv_{l-1}(p_{l-1}, q_0), ..., DRecv_0(q_0, q_0)) = Tmp$;
  endfor
endfor

Figure 44: Node program on processor $(p_{l-1}, ..., p_0)$ for multi-dimensional array redistribution.

characterization facilitates the packing and unpacking of array elements during array redistribution. Extensions of the developed closed forms to handle redistribution of multi-dimensional arrays are provided.
CHAPTER VI

Multi-Phase Array Redistribution

In this chapter, we address issues related to modeling and reducing the communication cost for array redistribution. Based on the closed forms developed for the processor and data index sets in Chapter V, we develop a communication and indexing cost model for two special forms of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem can be expressed in terms of these special forms. Using the closed form characterization a distributed scheduling algorithm which eliminates node contention for the all-to-many personalized communication for array redistribution is presented. The algorithm has an asymptotically lower communication and scheduling overhead than techniques presented in [107, 110]. Finally we present a multi-phase approach for reducing the communication cost of array redistribution. The underlying idea of the multi-phase approach is to perform array redistribution as a sequence of redistributions so that the total cost of the sequence is lower than the cost of the direct redistribution. Algorithms for determining the sequence of intermediate array distributions which minimizes the redistribution time are presented. Extensions of the algorithms to perform array redistributions involving multi-dimensional arrays are developed. Extensive experimental verification
on two state-of-the-art distributed-memory machines—the Cray T3D and the IBM SP2—demonstrate the efficacy of the multi-phase approach.

This chapter is organized as follows. A distributed scheduling algorithm and a cost model for the all-to-many personalized communication are developed in Section 6.1. In Section 6.2, an indexing cost model developed using the closed forms for the processor and data index sets developed in Chapter V is presented. Section 6.3 describes the multi-phase approach and provides algorithms for determining the sequence of intermediate distributions. Experimental results on the Cray T3D and IBM SP2 are presented in Section 6.4.

6.1 A Communication Cost Model for Array Redistribution

Array redistribution requires an all-to-many personalized communication, i.e., all processors will communicate with several other processors sending a distinct message to each destination processor. Modeling the communication cost for the all-to-many personalized communication requires accounting for the message startup and transmission costs and the overhead arising due to node and link contention. For wormhole routed and circuit-switched networks, the cost of transmitting a large message is nearly independent of the distance between the source and target nodes [94, 39].

Node contention arises as a processor can typically receive a single message at a time. Hence, if a target node has to receive messages from more than one source node at a time, then one of the source nodes has to wait to perform the communication. Link contention arises when two or more message paths need to share the same in-
terconnection network edge. The transmission of the messages in this case will be sequentialized (especially for circuit-switched networks).

Node contention can be avoided by scheduling the all-to-many personalized communication as a sequence of permutations. For several current architectures, with adaptive/randomized worm-hole routing the impact of node contention on the communication cost is more significant than that of link contention [109]. For such architectures, it is reasonable to model the cost of all-to-many personalized communication by factoring the communication into a sequence of permutations and determining the cost of each permutation.

In the absence of link contention, the cost of transmitting the largest message in a permutation gives the approximate cost of the permutation. If the message startup cost is $t_s$, the network bandwidth is $\frac{1}{c}$, and the largest message size in the permutation is $M$, then the cost of the permutation is approximately $t_s + Mt_e$. We first model the communication cost for redistribution of one-dimensional arrays and then extend the model to multi-dimensional arrays.

6.1.1 Array Redistribution of One-Dimensional Arrays

We model the cost of a redistribution from a cyclic($Yt$) distribution to a cyclic($t$) distribution. The cost for the cyclic($s$) to cyclic($Ys$) case can be similarly modeled.

*Cyclic($Yt$) to Cyclic($t$) Redistribution*

Using the send processor and data index sets developed in Theorem 5.1, the total number of message startups required and the volume of data to be communicated by
A processor $p$ is given by:

$$|P_{Send}(p)| = \min(P, Y), \quad (6.1)$$

$$|D_{Send}(p, q)| \approx \frac{N}{P \min(P, Y)}, \forall q \in P_{Send}(p). \quad (6.2)$$

A factoring of the all-to-many communication into permutations will contain at least $\min(P, Y)$ permutations. We show that for the cyclic($Yt$) to cyclic($t$) redistribution $\min(P, Y)$ is a tight lower bound on the number of permutations necessary to perform the communication for redistribution.

**Theorem 6.1** A tight lower bound on the number of permutations required to cover the communication for the cyclic($Yt$) to cyclic($t$) redistribution is $\min(P, Y)$.

**Proof:** Construct the $P \times P$ global communication matrix $COMM$ for the cyclic($Yt$) to cyclic($t$) redistribution, where $COMM(i, j) = 1$ if processor $i$ sends data to processor $j$, else $COMM(i, j) = 0$. From Theorem 5.1, it follows that each row and each column of $COMM$ has exactly $k = \min(P, Y)$ non-zero entries. From the Birkhoff-Von Neumann Theorem [14], we have

*If $M$ is an $m \times m$ matrix of nonnegative integers such that all the elements of each row and column add up to $k$ then $M$ is a sum of $k$ permutation matrices.*

Thus $COMM$ can be expressed as the sum of $\min(P, Y)$ permutation matrices. $\square$

Since the largest message size for each permutation is $\frac{N}{P \min(P, Y)}$, the cost of each permutation is $t_s + \frac{N}{P \min(P, Y)} t_e$. The total communication cost for the cyclic($Yt$) to
cyclic(t) redistribution is

\[
\min(P,Y) \cdot \left( t_s + \frac{N}{P \min(P,Y)} t_e \right) = \min(P,Y) t_s + \frac{N}{P} t_e.
\]  

(6.3)

While Theorem 6.1 demonstrates the existence of a tight lower bound, it does not provide an algorithm for scheduling the communication as a sequence of permutations to attain the lower bound. A graph theoretic framework based on expressing the communication as a bipartite graph and determining complete matchings of this graph [14, 40] can be used to determine the permutations. However, these schemes will require additional computation for construction of the communication matrix \(COMM\) and would incur the added expense of determining \(k\) complete matchings.

We develop a distributed routing algorithm for scheduling the communication which has \(\Theta(\min(P,Y))\) overhead. The algorithm uses the additional information obtained by the closed form characterization of the communication sets and does not require construction of the global communication matrix \(COMM\).
When \( Y \geq P \), the communication is a complete exchange which can be factorized into \( P \) permutations in a straightforward fashion. We focus on the case when \( Y < P \). From Theorem 5.1, when \( Y < P \), the first processor that \( p \) communicates with is \((pY) \mod P\). We refer to processors \( q \) such that \( q = (rY) \mod P \) for some \( 0 \leq r < P \) as first destination processors. After the first destination processor \((pY) \mod P\), processor \( p \) communicates with every consecutive processor (with a possible wrap-around at \((P - 1)\) up to and including \((pY + Y - 1) \mod P\). If each processor communicates with a distinct first processor, then clearly every consecutive processor it communicates with will be distinct and the all-to-many communication will be performed as a sequence of permutations. However it is possible that several processors communicate with the same first destination processor. For instance, consider an array \( A(96) \) distributed on eight processors using a cyclic(12) distribution and redistributed to a cyclic(2) distribution. The first processor that each processor communicates with is shown in Fig. 45. Lemma 6.1 shows that the processors can be partitioned into groups, such that all processors in a group communicate with the same first destination processor.

**Lemma 6.1** All processors \( p + \frac{k \cdot \text{lcm}(P,Y)}{Y} \), \( 0 \leq k < \gcd(P,Y) \), have the same first destination processor. The number of such distinct groups is \( \frac{\text{lcm}(P,Y)}{Y} \).

**Proof:** Follows by noting that \((p + \frac{k \cdot \text{lcm}(P,Y)}{Y}) \cdot Y \mod P = (pY + k \cdot \text{lcm}(P,Y)) \mod P = (pY) \mod P\). Each processor \( p \), \( 0 \leq p < \frac{\text{lcm}(P,Y)}{Y} \) determines a different group. \( \square \)

Let \( G(p) \) denote the group of processors \( p + \frac{k \cdot \text{lcm}(P,Y)}{Y} \) where \( 0 \leq k < \gcd(P,Y) \). There are totally \( \frac{\text{lcm}(P,Y)}{Y} \) such groups, each group identified by a \( p \in \left( 0 : \frac{\text{lcm}(P,Y)}{Y} - 1 \right) \).
From Lemma 6.1, it follows that the processors in \( G(p) \) communicate with the processors in \((pY \mod P : (pY \mod P + Y - 1) \mod P)\). Clearly, not all processors are first destination processors. Lemma 6.2 characterizes the first destination processors.

**Lemma 6.2** All processors \( p, 0 \leq p < P \) communicate first with a processor \( r \), such that \( \gcd(P, Y) \mid r \).

**Proof:** Follows by noting that a processor \( p \) first communicates with processor \( pY \mod P \) and \( pY \mod P = pY - kP = \gcd(P, Y)\left(pY - \frac{pY}{\gcd(P, Y)} - \frac{kP}{\gcd(P, Y)}\right) \) for some \( k \). \( \square \)

Thus processors which are not multiples of \( \gcd(P, Y) \) are not first destination processors. If we ensure that the processors in \( G(p) \) communicate with distinct first processors then the communication will be scheduled as a sequence of permutations. Since \( G(p) \) contains \( \gcd(P, Y) - 1 \) processors excluding \( p \) and the processors \(((pY \mod P + 1) \mod P : (pY \mod P + \gcd(P, Y) - 1) \mod P)\) are not first destination processors, by ensuring that processor \( p + k\frac{\text{lcm}(P, Y)}{Y} \) first communicates with processor \( pY \mod P + k \), we can guarantee that the communication is performed as a sequence of permutations.

The scheduling algorithm is shown in Fig. 46. In the algorithm, the send is non-blocking. The blocking receive is performed so as to receive any incoming message and the source processor \( p_r \) is determined from the received message. In Fig. 46, the first processor \( p_{sf} \) that \( p \) communicates with is determined as \( pY \mod P + \frac{pY}{\text{lcm}(P, Y)} \). As the computation within the loop is fixed, the complexity of the algorithm is \( \theta(\min(P, Y)) \).

The scheduling techniques for all-to-many personalized communication presented in [109] address a more general problem - scheduling the general irregular all-to-
Input: Processor index \( p \), total number of processors \( P \), source block size \( Yt \), target block size \( t \).

Output: Sequence of destination processors for scheduling all-to-many communication as sequence of permutations.

\[
\begin{align*}
\text{begin} \\
p_f &= p \ast Y \mod P; \\
p_l &= (p \ast Y + Y - 1) \mod P; \\
p_{sf} &= p_f + (p \ast Y) \div \text{lcm}(P, Y); \\
p_s &= p_{sf}; \\
\text{for } i = 0, \min(P, Y) - 1 \\
\quad &\text{Pack } A\_loc(DSend(p, p_s)) \text{ into } tmp\_sbuf; \\
\quad &\text{send}(p_s, tmp\_sbuf); \\
\quad &\text{recv}(p_r, tmp\_rbuf); \\
\quad &\text{Unpack } tmp\_rbuf \text{ into } B\_loc(DRecv(p, p_r)); \\
\quad &p_s = p_s + 1; \\
\quad &\text{if } (p_f \leq p_l \land p_s > p_l) \text{ then} \\
\quad &\quad p_s = p_f; \\
\quad &\text{else if } (p_f > p_l \land p_s = P) \text{ then} \\
\quad &\quad p_s = 0; \\
\quad &\text{else if } (p_f > p_l \land p_s = p_l + 1) \text{ then} \\
\quad &\quad p_s = p_f; \\
\quad &\text{endif} \\
\text{endfor}
\end{align*}
\]

Figure 46: Distributed algorithm for performing communication for \( \text{cyclic}(Yt) \) to \( \text{cyclic}(t) \) redistribution.
many personalized communication - and hence require the construction of the global communication matrix $COMM$ at runtime. Each processor then replicates the computation for the partitioning of $COMM$ into permutations. The scheduling overhead is $O(PY\log^2Y)$. Also for the all-to-many communication for the $cyclic(Yt)$ to $cyclic(t)$ redistribution, the techniques presented therein are not guaranteed to partition $COMM$ into exactly $\min(P,Y)$ permutations. The distributed scheduling algorithm in Fig. 46 has a scheduling complexity of $\Theta(\min(P,Y))$ and guarantees a partition into the minimum number of permutations.

$Cyclic(s)$ to $cyclic(Ys)$ **Array Redistribution**

The communication cost for the $cyclic(s)$ to $cyclic(Ys)$ redistribution can be similarly modeled. From Theorem 5.3, we have, for a processor $p$,

$$|PSend(p)| = \min(P,Y),$$

$$|DSend(p,q)| \approx \frac{N}{P\min(P,Y)} \forall q \in PSend(p).$$

Thus every processor communicates with $\min(P,Y)$ processors and the size of each message is approximately $\frac{N}{P\min(P,Y)}$. The communication cost of the array redistribution is determined to be equal to the cost in Eq. 6.3. Since the receive processor set for the $cyclic(s)$ to $cyclic(Ys)$ redistribution is equivalent to the send processor set for the $cyclic(Yt)$ to $cyclic(t)$ redistribution, the communication can be similarly partitioned into a sequence of $\min(P,Y)$ permutations using the algorithm in Fig. 46. Some additional computation cost is required for identification of the inverse. The scheduling overhead has a complexity of $O(P\min(P,Y))$. 
6.1.2 Array Redistribution of Multi-dimensional Arrays

For shape preserving multi-dimensional array redistribution, the communication cost estimates are obtained by combining the cost estimates for each dimension. The total number of message startups required and the volume of data communicated in a message is the product of the sizes of the processor send sets and data send sets for each dimension, respectively. This observation follows directly from Fig. 44. The distributed routing algorithm is obtained by applying the algorithm in Fig. 46 independently for each dimension to determine the next processor along that dimension to communicate with. Thus for an \( r \)-dimensional array \( A(0 : N_{r-1} - 1, \ldots, 0 : N_0 - 1) \) which is redistributed such that the redistribution along dimension \( i \) is either of the form \( \text{cyclic}(Y_i t_i) \) to \( \text{cyclic}(t_i) \) or \( \text{cyclic}(s_i) \) to \( \text{cyclic}(Y_i s_i) \) the communication cost for the redistribution is evaluated to be

\[
(min(P_{r-1}, Y_{r-1}) \times \cdots \times min(P_0, Y_0)) t_s + \left( \frac{N_{r-1} \times \cdots \times N_0}{P_{r-1} \times \cdots \times P_0} \right) t_e \tag{6.6}
\]

We now develop an indexing cost model for array redistribution.

6.2 An Indexing Cost Model for Array Redistribution

In this section, we develop a cost model for the indexing cost for array redistribution. The indexing overhead includes the cost of packing and unpacking the array elements into and from message buffers. This overhead includes a base cost for copying the entire local array into a buffer and back and an additional indexing cost for determining the elements to copy. The base cost is independent of the indexing strategy used and depends only on the processor architecture, while the additional indexing
/* Code for processor $p$ */

$p_f = (p * Y) \mod P$;
$p_i = (p * Y + Y - 1) \mod P$;

for $p_s = p_f, p_i$
    $y_1 = (p_s - p * Y) \mod P$;
    $l = y_1 * t$;
    $cnt = 0$;
    for $c = 0, C - 1$
        for $i = l, l + t - 1$
            $buffer[cnt] = A.loc[i]$;
            $cnt = cnt + 1$;
        endfor
    endfor
    $l = l + Y * t$;
endfor

Figure 47: Node code for packing data for array redistribution: node program for processor $p$.

cost is directly dependent on the goodness of the indexing scheme. We use the total number of contiguous sections of data elements to be packed into and unpacked from a message buffer as a measure of the additional indexing cost. Consider the sending phase of the redistribution from a $cyclic(Yt)$ to a $cyclic(t)$ distribution on a processor $p$ using the sets developed in Theorem 5.1 shown in Fig. 47.

Each contiguous section to be packed corresponds to an instance $(l, l + t - 1)$ of the inner-most loop. Clearly, a fixed cost is incurred for determining the lower bound $l$ for each section. Since consecutive elements are accessed, each section has good spatial locality and on an average a fixed cache penalty is incurred for each section.
Table 31: Indexing cost measures for special forms of array redistribution.

<table>
<thead>
<tr>
<th>Array Redistribution</th>
<th># Send Sections</th>
<th># Recv Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic(Yt) → cyclic(t)</td>
<td>( \frac{N}{Pyt} ) if ( Y &lt; P )</td>
<td>( \frac{N}{Psy} ) if ( Y &lt; P )</td>
</tr>
<tr>
<td></td>
<td>( \frac{N}{Pty} ) if ( Y \geq P )</td>
<td>( \frac{N}{Psy} ) if ( Y \geq P )</td>
</tr>
<tr>
<td>cyclic(s) → cyclic(Ys)</td>
<td>( \frac{N}{Py} ) if ( Y &lt; P )</td>
<td>( \frac{N}{Psy} ) if ( Y &lt; P )</td>
</tr>
<tr>
<td></td>
<td>( \frac{N}{Py} ) if ( Y \geq P )</td>
<td>( \frac{N}{Psy} ) if ( Y \geq P )</td>
</tr>
</tbody>
</table>

If the total number of sections to be packed/unpacked is \( S \), then the indexing cost would be approximately \( k \times S \), for a constant \( k \) whose value depends on the additional computation per section and the processor specifications.

The number of sections to be packed and unpacked for the redistribution from a \( cyclic(Yt) \) distribution to a \( cyclic(t) \) is determined from Theorems 5.1 and 5.2. A similar evaluation for the redistribution from a \( cyclic(s) \) to a \( cyclic(Ys) \) distribution can be performed using Theorems 5.3 and 5.4. These estimates are shown in Table 31.

The impact of the indexing cost on the total redistribution time will depend on the characteristics of the processor architecture and communication network. If the communication time dominates the redistribution time then the indexing time need not be accounted for in the cost model. The index cost for redistribution of multi-dimensional arrays can be similarly estimated.

6.3 A Multi-Phase Approach for Array Redistribution

In this section, we describe a multi-phase approach for performing array redistribution. We first consider the redistribution from a \( cyclic(Yt) \) to a \( cyclic(t) \) distribution
and then the general \textit{cyclic(s)} to \textit{cyclic(t)} redistribution. The redistribution from a \textit{cyclic(s)} to a \textit{cyclic(Ys)} distribution is similar to the \textit{cyclic(Yt)} to a \textit{cyclic(t)} case.

\subsection*{6.3.1 Array Redistribution from a Cyclic(Yt) to a Cyclic(t) Distribution}

Consider an array \(A(0:319)\) distributed onto 16 processors. A redistribution of \(A\) from a \textit{cyclic(20)} distribution to a \textit{cyclic(2)} distribution corresponds to the \textit{cyclic(Yt)} \(\rightarrow\) \textit{cyclic(t)} case with \(Y = 10\) and \(t = 2\). The communication cost of the redistribution is \(10t_s + 20t_e\). However, if the redistribution is performed in two phases, \textit{cyclic(20)} \(\rightarrow\) \textit{cyclic(10)} \(\rightarrow\) \textit{cyclic(2)}, the total communication cost is \((2t_s + 20t_e) + (5t_s + 20t_e) = 7t_s + 40t_e\). If the message startup cost \(t_s\) is significant compared to the element transmission cost \(t_e\) for the underlying architecture then the two-phase redistribution would require a smaller communication cost.

In general, the multi-phase strategy for redistribution of an array \(A(0: N-1)\) distributed onto \(P\) processors, from a \textit{cyclic(Yt)} to a \textit{cyclic(t)} distribution, corresponds to performing the redistribution as a sequence: \textit{cyclic(Yt)} \(\rightarrow\) \textit{cyclic(\(Y_1\t\))} \(\rightarrow\) \textit{cyclic(\(Y_1\times Y_2\t\))} \(\rightarrow\) \ldots \(\rightarrow\) \textit{cyclic(\(Y_1\times \cdots \times Y_{k-1}\times Y_k\t\))} \(\rightarrow\) \textit{cyclic(t)} where \(Y = Y_1\times \cdots \times Y_{k-1}\times Y_k\).

Phase \(i\) corresponds to a redistribution from a \textit{cyclic(\(Y_1\times \cdots \times Y_{i-1}\t\))} distribution to a \textit{cyclic(\(Y_1\times \cdots \times Y_{i-1}\t\))} distribution and has a communication cost of \(Y_{i+1}t_s + \frac{N}{P}t_e\). The total cost of the \(k\)-phase redistribution is given by

\[
\sum_{i=0}^{k-1} \left( Y_{i+1}t_s + \frac{N}{P}t_e \right) = \left( \sum_{i=1}^{k} Y_i \right) t_s + k\frac{N}{P}t_e.
\] (6.7)

Thus the multi-phase strategy trades off a decrease in the number of message startups with an increase in the data volume transmitted. Choosing the set of intermediate
data distributions is equivalent to choosing a multiplicative factorization \( Y = Y_1 \times \cdots \times Y_k \). The choice of the sequence of data distributions which has got the lowest execution time will depend on the actual values of the message startup time \( t_s \) and the data transmission time \( t_e \). We formulate the problem of determining the intermediate distributions as the following optimization problem:

Given the machine parameters \( t_s, t_e \) and \( P \), and the distribution parameters \( N \) and \( Y \), determine \( k \) and a factorization of \( Y = Y_1 \times \cdots \times Y_k \) which minimizes the total redistribution cost in Eq. 6.7.

For systems with a small number of processors, the total number of factorizations of \( Y \) is small, and an exhaustive search would be acceptable. However, for a large \( P \) and \( Y \) the development of algorithms or heuristics to solve the optimization problem is essential. When \( Y \) is a perfect power \( r^p \), finding a multiplicative partition of \( Y \) is equivalent to finding an additive partition of \( p \). We represent an additive partition of \( p \) by an \( l \)-tuple \((p_{l-1}, \ldots, p_0)\) where \( p = p_0 + \cdots + p_{l-1} \). The communication cost for this tuple is given by

\[
\left( \sum_{i=0}^{l-1} r^{p_i} \right) t_s + \frac{lN}{P} t_e. \tag{6.8}
\]

Finding the optimal solution corresponds to finding the tuple that minimizes the value of Eq. 6.8. The total number of tuples is given by the total number of additive partitions of \( p \), \( a(p) = \frac{1}{4\sqrt{3}^p} e^{\pi \sqrt{2/3} \sqrt{p}} \) [52] which is exponential. The optimization problem is solved in \( \theta(\log Y) \) time as follows using a method proposed in [99]. A minimum tuple of size \( l \) (as constructed below) is proved to have the the lowest value for Eq. 6.8
among all possible tuples of size \( l \) (Refer Appendix A for the proof). Thus among all tuples of size \( l \), only the value of Eq. 6.8 for the min-tuple need be considered. Let \( m(p, l) \) denote the min-tuple of size \( l \) for \( p \). We have

\[
m(p, l) = (\left\lfloor \frac{p}{l} \right\rfloor, \ldots, \left\lfloor \frac{p}{l} \right\rfloor, \left\lfloor \frac{p}{l} \right\rfloor, \ldots, \left\lfloor \frac{p}{l} \right\rfloor)
\]  

(6.9)

The algorithm generates min-tuples of sizes \( l, 1 \leq l \leq p \). The optimal tuple is found by comparing values of Eq. 6.8 for all the min-tuples in \( O(p) \) time. If \( Y \) is not a perfect power heuristics presented in Appendix B are used. The multi-phase approach for the cyclic(s) to cyclic(Ys) redistribution can be similarly developed.

### 6.3.2 Array Redistribution from a Cyclic(s) to a Cyclic(t) distribution

We now consider the general cyclic(s) to cyclic(t) redistribution where \( s \not\parallel t \) and \( t \not\parallel s \). This redistribution can be performed in two stages such that each of the two stages corresponds to a either a cyclic(s) to cyclic(Ys) redistribution or a cyclic(Yt) to cyclic(t) redistribution. The block size of the intermediate block-cyclic distribution is chosen as either a common multiple \( m \) or a common divisor \( d \) of \( s \) and \( t \). The choice of the divisor/multiple should be such that the total communication cost is reduced. A similar approach was proposed in [122].

Using the cost metrics developed for the special cases, the communication cost for the two-stage cyclic(s) \( \rightarrow \) cyclic(m) \( \rightarrow \) cyclic(t) distribution is \( \left( \frac{m}{s} t_s + \frac{N}{p} t_e \right) + \left( \frac{m}{t} t_s + \frac{N}{t} t_e \right) = m \left( \frac{1}{s} + \frac{1}{t} \right) t_s + 2 \frac{N}{p} t_e \). Clearly the choice of a multiple \( m \) which minimizes \( m \left( \frac{1}{s} + \frac{1}{t} \right) \) should provide the lowest communication cost. Since the least
common multiple $l = lcm(s, t)$ minimizes the expression, the two-stage redistribution using $cyclic(l)$ as the intermediate distribution should have the lowest communication cost. Similarly, the cost for the two-stage redistribution using a common divisor $d$ can be estimated as $\frac{1}{d}(s + t)t_s + 2\frac{t_t}{l}t_c$. It follows that the intermediate distribution $cyclic(g)$ where $g = gcd(s, t)$ should have the lowest communication cost among all common divisors of $s$ and $t$.

Since we are interested in finding the scheme which requires the lowest communication cost, a question that arises is which of the two-phase redistribution strategies has lower communication cost. As shown in Lemma 6.3, the common divisor and common multiple approach using the lcm and gcd, respectively, should have equal communication cost.

**Lemma 6.3** For $s, t \in \mathbb{Z}^+$, and $l = lcm(s, t)$, $g = gcd(s, t)$, $l \left( \frac{1}{s} + \frac{1}{t} \right) = \frac{1}{g}(s + t)$.

**Proof.** Since $st = lg$, we have $\frac{s}{g} + \frac{t}{g} = \frac{st}{gl} = \frac{st}{sl} = l \left( \frac{1}{s} + \frac{1}{t} \right)$. □

However, the indexing costs for the two schemes are different. The indexing costs for the two phases are composed in Table. 32.

For any common multiple $m$ and divisor $d$ of $s$ and $t$, we have $s \leq m$, $t \leq m$ and $s \geq d$, $t \geq d$. Thus from Table. 32 the common divisor approach will have greater indexing overhead than common multiple approach for any divisor $d$ and multiple $m$ which have equal communication costs. Thus the common multiple approach using $lcm(s, t)$ is expected to be the preferred two-phase scheme for performing redistribution from a $cyclic(s)$ to a $cyclic(t)$ distribution.
Table 32: Indexing cost measures for the two-phase array redistribution.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Multiple $m$</th>
<th>Divisor $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td># Send Sections</td>
<td>$Y_1 &lt; P \land Y_2 &lt; P$</td>
<td>$\frac{2N}{Pm}$</td>
</tr>
<tr>
<td></td>
<td>$Y_1 &lt; P \land Y_2 \geq P$</td>
<td>$\frac{N}{Pm} + \frac{2N}{P^d}$</td>
</tr>
<tr>
<td></td>
<td>$Y_1 \geq P \land Y_2 &lt; P$</td>
<td>$\frac{N}{P^d} + \frac{N}{Pm}$</td>
</tr>
<tr>
<td></td>
<td>$Y_1 \geq P \land Y_2 \geq P$</td>
<td>$\frac{N}{P^d}(\frac{1}{s} + \frac{1}{t})$</td>
</tr>
</tbody>
</table>

| # Recv. Sections                   | $Y_1 < P \land Y_2 < P$      | $\frac{2N}{Pm}$              |
|                                    | $Y_1 < P \land Y_2 \geq P$   | $\frac{N}{Pm} + \frac{2N}{P^d}$ |
|                                    | $Y_1 \geq P \land Y_2 < P$   | $\frac{N}{P^d} + \frac{N}{Pm}$ |
|                                    | $Y_1 \geq P \land Y_2 \geq P$| $\frac{N}{P^d}(\frac{1}{s} + \frac{1}{t})$ |

For common multiple $Y_1 = \frac{m}{s}$, $Y_2 = \frac{m}{t}$, For common divisor $Y_1 = \frac{1}{d}$, $Y_2 = \frac{s}{d}$

Once the common divisor/multiple has been identified, the multi-phase approach can be independently applied for each of the two stages. For example, the cyclic(36) → cyclic(80) redistribution can be performed as the two-stage sequence: cyclic(36) → cyclic(720) → cyclic(80). The first redistribution cyclic(36) → cyclic(720) corresponds to the cyclic(s) → cyclic(Ys) case while the second to the cyclic(Yt) → cyclic(t) case. The multi-phase approach can now be independently applied to the cyclic(36) → cyclic(720) and cyclic(720) → cyclic(80) redistributions performing them as cyclic(36) → cyclic(180) → cyclic(720) and cyclic(720) → cyclic(240) → cyclic(80), respectively. Using the multi-phase approach the number of message start-ups is halved while the data transmission cost is doubled.

The multi-phase approach can be similarly used for the redistribution of multi-
dimensional arrays. Consider a $t$-dimensional array $A(0 : N_{t-1} - 1, \ldots, 0 : N_0 - 1)$ distributed on a $P_{t-1} \times \cdots \times P_0$ virtual processor mesh. Consider a shape-preserving redistribution such that redistribution along dimension $i$ is either of the form $cyclic(Y_i t_i)$ to $cyclic(t_i)$ or $cyclic(s_i)$ to $cyclic(Y_i s_i)$. Let $Y = \min(P_{t-1}, Y_{t-1}) \times \cdots \times \min(P_0, Y_0)$. The communication cost of the redistribution is

$$Y_{ts} + \frac{N_{t-1} \times \cdots \times N_0}{P_{t-1} \times \cdots \times P_0} t_e.$$  \hfill (6.10)

The problem of determining the intermediate distributions in the multi-phase approach corresponds to finding an appropriate factorization of $Y$ which minimizes the total cost for the redistribution. The cost function is similar to that optimized in Eq. 6.7 and the optimization problem can be solved using the algorithms/heuristics for the single dimensional case.

### 6.4 Performance Results

In this section, we present experimental results to demonstrate the efficacy of the multi-phase approach for array redistribution. The experiments were performed on two state-of-the-art distributed-memory machines - a 32-node Cray T3D and a 64-node IBM SP2. The node programs on the Cray T3D used the PVM-message passing library, while the MPL message passing library with user space communication was used on the IBM SP2. Wall-clock times for redistributing arrays were measured on each machine, and the maximum finish time among all nodes reported.

We first evaluate the performance of the multi-phase approach for the $cyclic(Y t)$ to $cyclic(t)$ form of array redistribution. We compare the performance of the direct
Figure 48: Execution times for \textit{cyclic}(240) to \textit{cyclic}(8) and \textit{cyclic}(192) to \textit{cyclic}(8) redistributions on 32 node T3D.
redistribution with a two-phase and a three-phase redistribution. We then present similar measurements for the cyclic(s) to cyclic(Ys) form. The general cyclic(s) to cyclic(t) form of array redistribution is expressed in terms of these two special forms and an improved performance for either of the special forms directly implies better performance for the general form. For the cyclic(s) to cyclic(t) form of redistribution we compare the effect of choice of common multiples and divisors on the total redistribution time for the two-stage approach.

Fig. 48 presents the execution times on the Cray T3D for the cyclic(240) to cyclic(8) and the cyclic(192) to cyclic(8) redistributions of an array of double precision (eight-byte) numbers, respectively. Execution times for single-phase (1-P), two-phase (2-P), and three-phase (3-P) redistributions are presented. For the cyclic(240) to cyclic(8) redistribution, Y = 30, and the 2-P and the 3-P redistributions correspond to the factorizations: Y = 6 × 5 and Y = 2 × 3 × 5, respectively. For the cyclic(192) to cyclic(8) redistribution, Y = 24, and the 2-P and the 3-P redistributions correspond to the factorizations: Y = 6 × 4 and Y = 2 × 3 × 4, respectively. We observe that:

- The relationship between the rates of increase in execution time with array size is: 3-P > 2-P > 1-P. This follows by noting that for a fixed Y the component of the message startup time in the communication time is nearly fixed. An increase ΔN in the array size should lead to an increase of ΔNte in execution time for the 1-P redistribution, 2ΔNte for the 2-P redistribution, and 3ΔNte for the 3-P redistribution.
• For both redistributions and for all the array sizes considered, the 2-P redistribution performs better than the 3-P redistribution. This follows by noting that the Cray T3D has a low ts/te ratio. For the array sizes considered, a decrease of one message startup for the 3-P redistribution does not offset the increase in data transmission cost for the additional phase.

• For the cyclic(240) to cyclic(8) redistribution, the 2-P redistribution performs better than the 1-P redistribution up to an array size of approximately 100K. However, for the cyclic(192) to cyclic(8) redistribution the cross-over occurs at a smaller array size of 85K. This follows by noting that the reduction in the number of message startups for the 2-P strategy for Y = 30 is 19 while that for Y = 24 is 14. Hence the increase in data transmission cost will offset the reduction in startups at a smaller array size for Y = 24.

Experiments for different block sizes t, show a similar relationship between the one-phase, two-phase, and three-phase redistributions.

Fig. 49 presents the times on the IBM SP2 for the cyclic(3000) to cyclic(50) and the cyclic(1000) to cyclic(50) redistributions of an array of single precision (four-byte) numbers, respectively. For the cyclic(3000) to cyclic(50) redistribution, Y = 60, and the 2-P and the 3-P redistributions correspond to the factorizations: Y = 6 x 10 and Y = 5 x 4 x 3, respectively. For the cyclic(1000) to cyclic(50) redistribution, Y = 20, and the 2-P and the 3-P redistributions correspond to the factorizations Y = 5 x 4 and Y = 5 x 2 x 2, respectively. We observe the following:
Figure 49: Execution times for $cyclic(3000)$ to $cyclic(50)$ and $cyclic(1000)$ to $cyclic(50)$ redistribution on 64 node IBM SP2.
The relations between rates of increase in execution time with array size are:

3-P > 2-P > 1-P.

Clearly for \( Y = 20 \), the 3-P redistribution cannot have better performance than the 2-P redistribution for any array size as no reduction in message startups is obtained. For the \( \text{cyclic}(3000) \) to \( \text{cyclic}(50) \) case (\( Y = 60 \)), the 3-P redistribution has better performance than the 2-P redistribution up to an array size of 650K. This follows by noting that the IBM SP2 has a higher \( t_s/t_e \) ratio than the Cray T3D. For small array sizes the decrease in communication time due to a reduction in the number of message startups offsets the increase in data transmission cost due to an additional phase.

The cross-over between 2-P and 1-P redistributions occurs at an array size of 3.6M for the \( \text{cyclic}(3000) \) to \( \text{cyclic}(50) \) redistribution, while the cross-over for the \( \text{cyclic}(1000) \) to \( \text{cyclic}(50) \) occurs at a smaller array size of 1M. This difference arises since the 2-P redistribution for \( Y = 60 \) reduces the number of message startups by 44, while the 2-P redistribution for \( Y = 20 \) reduces the startups by 11. The 2-P redistribution performs better than the 1-P redistribution for a larger array size range on the SP2 than on the Cray T3D, due to the large \( t_s/t_e \) ratio.

Experiments for different block sizes \( t \), show a similar relationship between the one-phase, two-phase and three-phase redistributions.

The relationship between the one-phase, two-phase, and three-phase redistribution strategies for the \( \text{cyclic}(s) \) to \( \text{cyclic}(Ys) \) form of redistribution is similar to that for
Figure 50: Execution times for $\text{cyclic}(8)$ to $\text{cyclic}(240)$ and $\text{cyclic}(8)$ to $\text{cyclic}(192)$ redistribution on a 32 node T3D.
the cyclic(Yt) to cyclic(t) form of redistribution. Plots of the single and multi-phase strategies for a cyclic(8) to cyclic(240) and cyclic(8) to cyclic(192) redistribution on the Cray T3D and a cyclic(50) to cyclic(3000) and a cyclic(50) to cyclic(1000) on the IBM SP2, are shown in Fig. 50 and Fig. 51, respectively. The key-points to note are:

- On the Cray T3D, the cross-over between the 2-P and the 1-P redistributions occurs at an array size of 100K for the cyclic(8) to cyclic(240) redistribution, and at 87K for the cyclic(8) to cyclic(192) redistribution.

- On the IBM SP2, the cross-over between the 2-P and the 1-P redistributions occurs at an array size of 4M for the cyclic(50) to cyclic(3000) redistribution, and at 1M for the cyclic(8) to cyclic(192) redistribution. The cross-over between the 2-P and 3-P occurs at an array size of 600K for the cyclic(50) to cyclic(3000) redistribution. For the cyclic(50) to cyclic(1000) redistribution, the 2-P redistribution performs better than the 3-P redistribution for all array sizes considered.

We now evaluate the effect of the choice of common multiples and divisors on the general cyclic(s) to cyclic(t) redistribution. Table. 33 shows the two-stage redistribution times for various source and target block sizes on the Cray T3D. The redistribution times for various multiples and divisors are presented. It can be observed that the two-stage redistributions using the lcm l and the gcd g as block sizes for the intermediate distribution perform better than those using a common multiple greater than the lcm and a common divisor smaller than the gcd. The relationship between the two-stage strategies for multiples is: \( t_1 < t_{i2} < t_{i4} \). Similarly, for the
Figure 51: Execution times for \textit{cyclic}(50) to \textit{cyclic}(3000) and \textit{cyclic}(50) to \textit{cyclic}(1000) redistribution on 64 node IBM SP2.
Table 33: Execution times (ms) for \textit{cyclic(s)} to \textit{cyclic(t)} redistribution on 32-processor Cray T3D.

<table>
<thead>
<tr>
<th>$N = 552960$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$t$</td>
<td>$lcm$</td>
<td>$lcm \times 2$</td>
<td>$lcm \times 4$</td>
<td>$gcd$</td>
<td>$gcd/2$</td>
</tr>
<tr>
<td>120</td>
<td>180</td>
<td>22.4</td>
<td>24.2</td>
<td>27.2</td>
<td>22.5</td>
<td>24.7</td>
</tr>
<tr>
<td>120</td>
<td>540</td>
<td>24.2</td>
<td>27.7</td>
<td>34.5</td>
<td>24.5</td>
<td>28.3</td>
</tr>
<tr>
<td>240</td>
<td>180</td>
<td>23.1</td>
<td>25.3</td>
<td>29.4</td>
<td>23.4</td>
<td>25.7</td>
</tr>
<tr>
<td>240</td>
<td>540</td>
<td>24.9</td>
<td>28.7</td>
<td>36.5</td>
<td>25.2</td>
<td>29.3</td>
</tr>
<tr>
<td>480</td>
<td>180</td>
<td>24.2</td>
<td>27.4</td>
<td>34.0</td>
<td>24.2</td>
<td>28.0</td>
</tr>
<tr>
<td>480</td>
<td>540</td>
<td>25.9</td>
<td>31.0</td>
<td>40.9</td>
<td>26.2</td>
<td>31.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$N = 2211840$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$t$</td>
<td>$lcm$</td>
<td>$lcm \times 2$</td>
<td>$lcm \times 4$</td>
<td>$gcd$</td>
<td>$gcd/2$</td>
</tr>
<tr>
<td>120</td>
<td>180</td>
<td>84.6</td>
<td>87.5</td>
<td>91.1</td>
<td>85.3</td>
<td>89.8</td>
</tr>
<tr>
<td>120</td>
<td>540</td>
<td>86.4</td>
<td>91.0</td>
<td>105.9</td>
<td>87.5</td>
<td>93.3</td>
</tr>
<tr>
<td>240</td>
<td>180</td>
<td>86.0</td>
<td>88.2</td>
<td>92.5</td>
<td>86.6</td>
<td>90.5</td>
</tr>
<tr>
<td>240</td>
<td>540</td>
<td>87.7</td>
<td>91.7</td>
<td>107.0</td>
<td>88.8</td>
<td>94.0</td>
</tr>
<tr>
<td>480</td>
<td>180</td>
<td>86.5</td>
<td>89.9</td>
<td>96.7</td>
<td>87.3</td>
<td>92.1</td>
</tr>
<tr>
<td>480</td>
<td>540</td>
<td>88.3</td>
<td>93.5</td>
<td>110.3</td>
<td>89.5</td>
<td>95.6</td>
</tr>
</tbody>
</table>
divisors we have: $t_g < t_{g/2} < t_{g/4}$. These relations can be explained by noting that the two-stage redistribution using either the lcm or the gcd as the block size of the intermediate block-cyclic distribution has the lowest communication cost among all common divisors and multiples, as discussed in Section 6.3. Furthermore it can be observed that $t_{l*2} < t_{g/2}$ and $t_{l*4} < t_{g/4}$. While the two-stage methods using the lcm and the gcd have nearly equal performance for a small array size, for a larger array size $t_l < t_g$. Although these schemes are predicted to have the same communication cost, a two-stage redistribution using a multiple will have a lower indexing cost than the redistribution using a divisor as is explained by the indexing cost model presented in Section 6.2.

Execution times for the \textit{cyclic(s)} to \textit{cyclic(t)} form of array redistribution on the IBM SP2 are presented in Table 34. Similar relationships between various multiples and divisors are observed.

### 6.5 Summary

In this chapter, we have developed a distributed scheduling algorithm, an indexing cost model, and a communication cost model for the all-to-many personalized communication arising due to two special forms of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem can be expressed in terms of these special cases. The algorithm was developed using the closed forms for the processor index sets developed in Chapter V. The schedule generated by the algorithm eliminates node contention and has a lower communication and
Table 34: Execution times (ms) for \textit{cyclic}(s) to \textit{cyclic}(t) redistribution on 64-processor IBM SP2.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{N} = 110590 & & & & & & & & \\
\hline
\textbf{s} & \textbf{t} & \textbf{l} & \textbf{l*2} & \textbf{l*4} & \textbf{g} & \textbf{g/2} & \textbf{g/4} \\
\hline
120 & 180 & 317 & 333 & 341 & 320 & 324 & 343 \\
\hline
120 & 540 & 304 & 478 & 707 & 314 & 487 & 715 \\
\hline
240 & 180 & 351 & 281 & 462 & 366 & 271 & 448 \\
\hline
240 & 540 & 344 & 424 & 798 & 360 & 431 & 806 \\
\hline
480 & 180 & 292 & 374 & 696 & 292 & 370 & 738 \\
\hline
480 & 540 & 289 & 521 & 1048 & 308 & 558 & 1055 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{N} = 4423680 & & & & & & & & \\
\hline
\textbf{s} & \textbf{t} & \textbf{l} & \textbf{l*2} & \textbf{l*4} & \textbf{g} & \textbf{g/2} & \textbf{g/4} \\
\hline
120 & 180 & 759 & 1013 & 1115 & 757 & 1040 & 1137 \\
\hline
120 & 540 & 886 & 1126 & 1155 & 889 & 1159 & 1182 \\
\hline
240 & 180 & 847 & 1126 & 1324 & 925 & 1133 & 1361 \\
\hline
240 & 540 & 1022 & 1208 & 1344 & 1047 & 1236 & 1416 \\
\hline
480 & 180 & 960 & 1265 & 1113 & 963 & 1363 & 1231 \\
\hline
480 & 540 & 706 & 772 & 849 & 732 & 813 & 881 \\
\hline
\end{tabular}
\end{table}
scheduling overhead than algorithms presented in the literature. A multi-phase approach for reducing the communication cost of array redistribution was presented. Algorithms for identifying the sequence of intermediate distributions were presented. Experimental results on the Cray T3D and the IBM SP2 demonstrate the validity of the developed cost models and the efficacy of the multi-phase approach.
CHAPTER VII

Conclusions

In this thesis, we have developed compile-time and run-time methods for the efficient execution of array statements and for performing array redistribution on distributed-memory machines.

7.1 Summary of Principal Contributions

We have presented techniques for efficient enumeration of the data index sets and processor index sets for array statement execution involving arrays distributed using the block, cyclic, and block-cyclic distributions. Closed forms expressed as regular sections for determining the send and receive data and processor sets for block and cyclic distributions were derived. A strategy based on virtual processors was used along with these closed forms to generate efficient indexing code for arrays distributed using block-cyclic distributions. A heuristic for selecting the appropriate virtual views at the source and target distribution was developed. Performance results for an implementation on the Cray T3D were presented. The performance results demonstrate that the developed heuristic provides a good indication of the virtualization scheme with the lowest execution time. It is also observed that for a majority of the cases
considered, the table generation overhead for the virtual processor approach is a small percentage of the execution time.

We have addressed the memory allocation and index set enumeration problem for arrays mapped using a two-level mapping with non-unit stride to a set of processors. Using simple mathematical properties of regular sections, we identified the regular section of array elements mapped to a processor for block and cyclically distributed arrays. This regular section characterization identifies the exact amount of memory required on each processor for block and cyclically distributed arrays. Furthermore, it facilitates the index set translation from the local template space to the compressed array space. We have identified methods for extending these techniques to the virtual processor approach for handling block-cyclically distributed arrays. Extensive experimental evaluation on the Cray T3D demonstrates that while the scheme with hole allocation will require additional computation at compile time, it leads to significant savings in memory used. Furthermore, while the table generation times for the execution of the array statement increases, the time for execution of the array statement is reduced due to an improvement in spatial locality.

We have also presented a multi-phase approach for performing communication-efficient data redistribution for block-cyclically distributed arrays. Precise closed form expressions have been developed for the send and receive processor and data index sets for two special cases where the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. These closed forms facilitate the development of a distributed scheduling algorithm.
for performing the all-to-many personalized communication, and a communication and indexing cost model for array redistribution. The algorithm eliminates node contention, incurs minimum communication overhead and has an asymptotically lower complexity than the algorithms presented in the literature. Based on this model, a multi-phase approach for performing the all-to-many communication for array redistribution is proposed. The key idea of the multi-phase approach is to perform the array redistribution as a sequence of redistributions such that the cost of the sequence is lower than that of the direct redistribution. Performance results on the Cray T3D and the IBM SP2 demonstrate that the multi-phase strategy can improve performance over the single phase strategy for array redistribution.

7.2 Directions for Future Research

In this research, we have developed methods for efficiently executing array statements and for performing array redistribution on distributed-memory machines. This work can be extended along several directions.

- Incremental index set generation: Typically, several array statements in HPF programs involve arrays which are identically distributed and array sections which are similar to those in previously executed statements. Techniques for identifying situations under which the previously evaluated sets can be reused, or efficient methods used for incremental evaluation of the index sets, are important for reducing the run-time overhead of array statement execution. Preliminary results along this direction have been presented in [76].
- Modeling link contention: Scheduling techniques for the all-to-many personalized communication have received relatively little attention. For fixed routing networks such as the iPSC/860, the impact of link contention on the total communication cost is significant. For such architectures, modeling the link contention arising due to the all-to-many personalized communication arising during redistribution is important. Also, methods for scheduling the communication such that the least amount of link contention is incurred would be required.

- Scheduling general irregular communication: Few methods have been presented for scheduling the irregular all-to-many personalized communication for the general array statement. Graph theoretic frameworks based on partitioning the communication graph for scheduling the communication to reduce the node contention can be developed.
APPENDIX A

Proof of Optimality of Min-Tuple for $Y = r^k$

We now prove the optimality of the min-tuple presented in Section 6.3. We first formally define the min-tuple. Let $\mathcal{T}(k, l)$ denote the set of all ordered $l$-tuples corresponding to additive partitions of $k$. An ordered $l$-tuple $(k_{l-1}, \ldots, k_0)$ in $\mathcal{T}(k, l)$ satisfies the relation that $\sum_{i=0}^{l-1} k_i = k$ and $k_i \geq k_{i-1}$, $1 \leq i < l$. Given a tuple $s \in \mathcal{T}(k, l)$, let $s_i$ represent component $i$ of $s$. The communication cost for the ordered $l$-tuple $s = (s_{l-1}, \ldots, s_0)$ as given by Eq. 6.7 is $C(s) = (\sum_{i=0}^{l-1} r^{s_i}) \cdot t_s + l \cdot \frac{N}{p} t_e$.

We now define some relations between two tuples in $\mathcal{T}(k, l)$ and based on this relation impose a linear irreflexive order on $\mathcal{T}(k, l)$.

**Definition A.1** Let $c \in [0 : l - 1]$; then for any two ordered tuples $r, s \in \mathcal{T}(k, l)$, $r \prec_c s \iff (\forall j \in [c + 1 : l - 1] : r_j = s_j) \land r_c < s_c)$.

**Lemma A.1** Given the set of tuples $\mathcal{T}(k, l)$ we have.

1. $\forall c \in [0 : l - 1]$, $\prec_c$ is an irreflexive partial order on $\mathcal{T}(k, l)$.

2. $\forall r, s \in \mathcal{T}(k, l)$, either $r = s$ or there exists a unique $c \in [0 : l - 1]$ such that either $r \prec_c s$ or $s \prec_c r$.  

193
3. \( \forall r, s, u \in \mathcal{T}(k, l) \) and \( c, c' \in [0 : l - 1] \), \( r \prec_c s \wedge s \prec_c u \Rightarrow r \prec_{c''} u \) where 
\( c'' = \max(c, c') \).

We now define the lexicographic relation \( \prec \) between two ordered \( l \)-tuples.

**Definition A.2 (Lexicographic Order)** \( \forall r, s \in \mathcal{T}(k, l) : r \prec s \iff \exists c \in [0 : l - 1] : r \prec_c s \).

The relation \( \prec \) on \( \mathcal{T}(k, l) \) is similar to the lexicographical order \( < \) defined on \( \mathbb{Z}^n \).

The relations \( \leq, \succ \), and \( \succeq \) are defined equivalently. Using Lemma A.1 and Def. A.2, we have:

**Lemma A.2** The lexicographical order \( \prec \) on \( \mathcal{T}(k, l) \) is an irreflexive linear order.

From Lemma A.2 and noting that the set \( \mathcal{T}(k, l) \) is finite, it is evident that \( \mathcal{T}(k, l) \) has a unique minimum. We refer to the minimum as the min-tuple, and denote it as \( m(k, l) \). Noting that \( k = \left\lceil \frac{k}{l} \right\rceil (k \mod l) + \left\lfloor \frac{k}{l} \right\rfloor (l - k \mod l) \) the min-tuple can be proven to be.

\[
m(k, l) = \left( \left\lfloor \frac{k}{l} \right\rfloor, \ldots, \left\lfloor \frac{k}{l} \right\rfloor, \left\lceil \frac{k}{l} \right\rceil, \ldots, \left\lceil \frac{k}{l} \right\rceil \right)
\]  
(A.1)

The proof follows by noting that

- Any tuple \( s \) such that \( s_{l-1} > \left\lceil \frac{k}{l} \right\rceil \) is greater than the min-tuple and
- The min-tuple is the only tuple with largest component equal to \( \left\lceil \frac{k}{l} \right\rceil \).

The values of the components of the min-tuple have a characteristic property, i.e., there is at most one "jump" of size one in the component values. For instance, the
min-tuple \( m(44, 8) = (6, 6, 6, 5, 5, 5, 5) \) can be represented as shown in Fig. 52. The area under the solid-line curve is 44. This property is stated in Lemma A.3 and follows directly from the definition of the min-tuple and noting that \( \lceil \frac{k}{I} \rceil - \left\lfloor \frac{k}{I} \right\rfloor \leq 1 \).

**Lemma A.3** \( 0 \leq m(k, l)_i - m(k, l)_{i-1} \leq 1, \quad 0 < i < l - 1 \)

This property of the min-tuple leads to the following lemma.

**Lemma A.4** Let \( t \in T(k, l) \) be any tuple \( t \neq m(k, l) \) and let \( c \in [0 : l - 1] \) be the largest index such that \( m(k, l)_c \geq t_c \). Then \( m(k, l)_i \leq t_i, \quad c < i < l \) and \( m(k, l)_i \geq t_i, \quad 0 \leq i \leq c \).

The proof follows by considering the three possible relations between the tuple \( t \) and \( m(k, l) \) as shown in Fig. 53. The lemma simply states that once the plot for the tuple \( t \) goes below that of the mintuple, it remains below. Using Lemma A.4 we prove that the min-tuple \( m(k, l) \) has the lowest cost in \( T(k, l) \). We first introduce the concept of adjacent tuples.

**Definition A.3 (Adjacent Tuples)** Two ordered tuples \( r, s \in T(k, l) \) are adjacent if they differ in exactly two components, i.e., \( r_{c1} \neq s_{c1}, r_{c2} \neq s_{c2}, \) and \( r_i = s_i, \forall i, \ i \neq c_1, \ i \neq c_2 \).
Note that the adjacent tuples have to differ in exactly two components and that $r_{c_1} - s_{c_1} = s_{c_2} - r_{c_2}$, where $c_1$ and $c_2$ are the indices in which $r$ and $s$ differ. The relation between the costs of two adjacent tuples is provided in the following lemma.

**Lemma A.5** If $r, s \in \mathcal{T}(k, l)$ are adjacent tuples, then $r < s \Rightarrow C(r) < C(s)$.

**Proof:** Since $r < s$ and $r, s$ are adjacent, it follows that $r$ and $s$ are of the form $r = (x_{l-1}, \ldots, x_p, \ldots, x_q, \ldots, x_0)$ and $s = (x_{l-1}, \ldots, x_p + c, \ldots, x_q - c, \ldots, x_0)$, where $c > 0$. Consider $C(s) - C(r)$. We have $C(s) - C(r) = r_{x_p+c} - r_{x_q-c} - r_{x_p} + r_{x_q} = r_{x_p}(r^c - 1) - r_{x_q}^{-c}(r^c - 1) = (r^c - 1)(r_{x_p} - r_{x_q}^{-c})$. Since $x_p \geq x_q$ and $c > 0$ we have $C(s) - C(r) > 0$. □.

Given any tuple $t \in \mathcal{T}(k, l)$, it is possible to construct a chain of adjacent tuples starting at $t$ and ending in the min-tuple $m(k, l)$. We have the following lemma.

**Lemma A.6** Given a tuple $t \in \mathcal{T}(k, l)$, there exists a chain of adjacent tuples $t, t_1, t_2, \ldots, t_r, m(k, l)$ in $\mathcal{T}(k, l)$, such that $t \geq t_1, t_i \geq t_{i+1}, 1 \leq i < r - 1$, and $t_r \geq m(k, l)$.
For example, the chain for the tuple $(9, 7, 6, 5, 4, 2) \in \mathcal{T}(33, 6)$ is: $(9, 7, 6, 5, 4, 2)$, $(9, 6, 6, 5, 5, 2)$, $(6, 6, 6, 5, 5, 5)$. The chain can be constructed by iteratively constructing $t_{i+1}$ from $t_i$ and $m(k, l)$ using Lemma A.4. Thus we have

**Theorem A.1** The min-tuple $m(k, l)$ has the lowest cost among all tuples $t \in \mathcal{T}(k, l)$.

**Proof:** Follows from Lemma A.5 and Lemma A.6. \qed
Heuristics for General $Y$

We consider different cases of the general $Y$. Depending on the relationship between $t_s$ and $t_e$ we can identify optimal solutions or use heuristics.

We first consider the case when $t_s >> \frac{N}{P} t_e$. This case occurs when redistributing small arrays or for architectures with large message setup times. The cost function in Eq. 6.7 is

$$\sum_{i=0}^{k-1} \left( Y_{i+1} t_s + \frac{N}{P} t_e \right) \approx \sum_{i=0}^{k-1} (Y_{i+1}) t_s$$ (B.1)

Thus we are interested in finding a factorization of $Y = Y_1 \times \cdots \times Y_k$ which minimizes $\sum_{i=1}^{k} Y_i$. We have $\forall r, s \in \mathbb{Z}, rs \geq r + s$. It follows that the prime factorization of $Y$ will have the lowest communication time. This case is at one extreme at which the data transmission cost is negligible compared to the setup cost. The other extreme corresponds to the case when the setup cost is negligible. For this case the single phase redistribution will have lowest cost.

For the other cases, we use a greedy heuristic to minimize the objective function in Eq. 6.7. The heuristic starts with the prime factorization of $Y$. At each stage the algorithm combines a pair of factors, such that this grouping leads to the largest
reduction in the objective function. If $Y$ is composed of $t$ prime factors each step requires $O(t^2)$ steps, as at most $t^2$ pairs have to be examined for combining. This grouping is continued until no further reduction in the value of the objective function is possible. The grouping will terminate in at most $t - 1$ steps. Thus, the heuristic has a $O(t^3)$ complexity. If no reduction is possible in the first step then the prime factorization is the optimal. However, in the general case the heuristic may get into a local minima and may not yield the optimal solution.
BIBLIOGRAPHY


[125] C.-W. Tseng. *An Optimizing Fortran-D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Department of Computer Science, Rice University, P.O. Box, Houston, TX 77251, Jan. 1993. also available as RICE COMP TR93-199.


