Stellingen
behorende bij het proefschrift

Design of a HPF compiler
A compilation framework for a data-parallel language

W.J.A. Denissen

1. Om echt een hoge performance te bereiken op een data parallelle machine dient men zich niet alleen op het efficiënt communiceren van data te concentreren maar ook op het meest elementaire; het efficiënt opslaan en enumereren van deze data.

2. Een nieuw programmeermodel kan niet alle problemen oplossen waar oudere programmeermodellen dat niet konden. Het besef dat ieder programmeermodel zijn eigen gebied kent waarin deze excelleert is nog niet bij iedereen doorgedrongen.

3. Programmeertalen met krachtige staalconstructies zijn moeilijker te optimaliseren dan programmeertalen met minder krachtige taalconstructies. Het duidelijkheid komt dit naar voren in het efficiëntieverschil van de oude FORTRAN 77 compilers vergeleken met de nieuwe FORTRAN 90 compilers.

4. Uit bedrijfsstrategisch oogpunt is bij dataparallele rekenintensieve applicaties HPF te prefereren boven het directe gebruik van message passing. HPF garandeert een goede portabiliteit en minimaliseert de onderhoudskosten.

5. Voorkomen is beter dan genezen, dat geldt ook voor het programmeerproces. Door bijvoorbeeld gebruik te maken van syntax directed editors, kunnen inconsistentie specificaties gemakkelijk vooraf voorkomen worden.

6. De Atlantische Oceaan werd al vroeg in de bronstijd en misschien zelfs daarvoor al bevaren door de mens, gegeven de volgende feiten:

   • “De Odyssee is onder meer een mondelinge kaart van de Atlantische Oceaan geweest, met nauwkeurige instructies voor de Keltische zeevaarders”. Uit “Waar eens Troje lag” van Iman Wilkens.

   • Er cocainresten gevonden zijn in sommige faraomummies die alleen hun oorsprong in Zuid-Amerika kunnen hebben gehad.

   • In Midden-Amerika zijn uit dezelfde periode beelden gevonden van personen met Afrikaanse c.q. Europese gelaatstrekken.

   • Thor Heyerdahl heeft aangetoond dat met een papyrusboot een oversteek te maken is van Afrika naar Zuid-Amerika, volgens een route zoals beschreven in de Odyssee.
7. Indien de overheid de veiligheid van de weggebruikers voorstelt, dient er niet bestaat te worden op te snel rijden maar op onverantwoord rijden. Een maat voor onverantwoord rijden is een combinatie van de verschil snelheid t.o.v. de weg EN t.o.v. de medeweggebruikers. Waarbij dat laatste misschien nog wel het belangrijkste is.

8. In de 20e eeuw gold; tijd is geld. Voor de 21e eeuw zal gelden; tijd is waardevol.

9. De veelgebezigde term “time to market” vertaalt zich vaak in “haastige spoed is zelden goed”. De haastige spoed is enkel goed voor de kranten die de terughaal advertenties mogen plaatsen.

10. Een autorit met de sollicitant geeft meer inzicht dan een goed gesprek.
Design of an HPF compiler
A compilation framework for a data-parallel language

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. F. Wakker,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 5 september 2000 om 10.30 uur
door Wilhelmus Josephus Adrianus DENISSEN
elektrotechnisch ingenieur
geboren te Berkel-Enschot.
Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. ir. H. J. Sips

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof. dr. ir. H. J. Sips, Technische Universiteit Delft, promotor
Prof. dr. ir. J. van Katwijk, Technische Universiteit Delft
Prof. dr. ir. H. E. Bal, Vrije Universiteit Amsterdam
Prof. dr. H. A. G. Wijshoff, Universiteit Leiden
Prof. dr. ir. W. Lourens, Rijks Universiteit Utrecht
Dr. ir. H. X. Lin, Technische Universiteit Delft
Dr. T. Brandes, GMD-SCAI, St. Augustin, Duitsland

ISBN 90-6464-197-8

Copyright © 2000 by W. J. A. Denissen

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.
Aan mijn ouders
Acknowledgments

In mid 1992, I first encountered the challenges of compiler design when I started to work on the European ESPRIT-project PREPARE. Thanks to Edwin Paalvast who introduced me to this project. At the time, we were both working at the Institute for Applied Information Technology of the Netherlands Organization for Applied Scientific Research (TNO-ITI). The PREPARE project aimed at building a programming environment for a parallel machine.

During the project, I learned more about managing the different aspects of a large project. Martijn de Lange CEO of ACE showed me the art of how to maneuver the project members into the direction of the ever changing targets. Arthur Veen, as the technical director of the project, responsible for the technical realization and coherence of the different deliverables, taught me the balance between enthusiasm and realization as scheduled. He also showed that his hobby of being an actor also helped to achieve its goals.

For the technical support, I would like to thank Hans van Someren and Erik van Konijnenburg of ACE, Olivier Cheron, Jean-Louis Pazat and Francois Andre of Irisa, and Peter Brezany of the University of Vienna. They were a great help in the many discussions about how to model data distributions.

The national successor project FUTURE focused at the proliferation of data-parallel compiler technology among the research institutes and educational infrastructure. Together with Rob Kurver of ACE I learned that market considerations, politics, and sometimes just plain luck can have a much larger effect on the success of a product than maturity, robustness, and efficiency of the developed technology.

I specially want to thank my promoter Henk Sips who has introduced me into the small and exclusive world of compiler developers in the United States and Europe. I really enjoyed the open atmosphere and pleasant competition in the discussions between the compiler experts. In numerous occasions, the tray-tables of the airplanes were far to small to write down our ideas of how to extend our compilation model. The long times that it took to travel, were sometimes not enough to complete the discussions.

I would like to thank Kees van Reeuwijk and Vincent Korstanje for their critical feedback in our brainstorm sessions. These lively discussions were sometimes both at a high technical level and at a high audio level. So, I also want to thank my colleges at TPD working the offices in the same corridor, for not complaining. Kees van Reeuwijk helped me a lot in properly describing the methods and principles we had found. Vincent Korstanje did a lot of benchmarking of our PRECOMPILER compiler, thereby finding a lot of bugs or shortcomings in the implementation of the compiler.

Without the support of my employer the Institute of Applied Physics of the Netherlands Organization for Applied Scientific Research (TNO-TPD), I could never have completed this thesis.
ACKNOWLEDGMENTS

Especially Koos van Woerden who made it possible to reserve some resources (mainly time), and Peter Maarleveld who managed to fit in my Ph.D. work into my other obligations.

I would like to thank my family and especially my wife Jeske and my son Thijs and my daughter Fleur for their support, in deviating my mind from compiler topics to other joyful areas. Finally, I would like to thank my parents who gave me the opportunity and support to get me where I am now.
# Table of Contents

ACKNOWLEDGMENTS

1 INTRODUCTION
   1.1 History of the Research
   1.2 Motivation of the Research
   1.3 The Research Question
   1.4 An Outline of This Thesis
   1.5 Contributions of This Research

2 Compiler Targets
   2.1 Shared-Memory Machines
   2.2 Distributed-Memory Machines
   2.3 Parallel Abstract Machine (PAM)
   2.4 Single Program Multiple Data Execution Model (SPMD)

3 High Performance Fortran (HPF)
   3.1 History
   3.2 Data Distribution Directives
      3.2.1 The Processors directive
      3.2.2 The Template directive
      3.2.3 The Distribute directive
      3.2.4 The Align directive
      3.2.5 The Inherit directive
   3.3 Work Distribution Directives
      3.3.1 The On directive
   3.4 Kinds of Assignments
      3.4.1 The Independent directive

4 The Compilation Model
   4.1 Terminology
   4.2 Normalization
      4.2.1 Normalizing HPF data mappings
      4.2.2 Normalized volume assignment
   4.3 The Owner Concept
      4.3.1 Work distribution
      4.3.2 Iteration Space distribution
      4.3.3 Mapping application and mapping extraction
   4.4 The Compilation System (CoSY)
      4.4.1 The HPF Front-end
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2</td>
<td>The Align engine</td>
<td>40</td>
</tr>
<tr>
<td>4.4.3</td>
<td>The Decompose engine</td>
<td>41</td>
</tr>
<tr>
<td>4.4.4</td>
<td>The SPMD engine</td>
<td>43</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Parallel IR Extensions</td>
<td>44</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Parallel Engines Run Time System (PERTS)</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>ALIGNMENT ANALYSIS</td>
<td>48</td>
</tr>
<tr>
<td>4.5.1</td>
<td>The alignment analysis phase</td>
<td>50</td>
</tr>
<tr>
<td>4.5.2</td>
<td>The alignment selection phase</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>LOWERING BASIC ASSIGNMENTS</td>
<td>55</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Assignments and communication</td>
<td>55</td>
</tr>
<tr>
<td>4.6.2</td>
<td>More about locations</td>
<td>57</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Lowering Scalar assignments</td>
<td>58</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Lowering Volume assignments</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>RELATED WORK</td>
<td>65</td>
</tr>
<tr>
<td>4.7.1</td>
<td>The Owner Concept</td>
<td>65</td>
</tr>
<tr>
<td>4.7.2</td>
<td>The Compilation System</td>
<td>66</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Related compiler developments</td>
<td>66</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Alignment Analysis</td>
<td>67</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Basic Assignments</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>STORAGE AND ITERATION SET GENERATION</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>STORAGE</td>
<td>69</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Allocation of mapped arrays</td>
<td>69</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Send and receive buffer management</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>ENUMERATION</td>
<td>74</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Code generation</td>
<td>74</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Polyhedron scanning</td>
<td>77</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Iteration set intersections</td>
<td>80</td>
</tr>
<tr>
<td>5.3</td>
<td>RELATED WORK</td>
<td>82</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Storage</td>
<td>82</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Enumeration</td>
<td>83</td>
</tr>
<tr>
<td>6</td>
<td>OPTIMIZATIONS</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>PRE-SPMD OPTIMIZATIONS</td>
<td>87</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Message merging</td>
<td>87</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Hoisting assignments</td>
<td>89</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Squeezing independent do loops</td>
<td>90</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Stencil references</td>
<td>90</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Reductions</td>
<td>92</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Masked assignments</td>
<td>94</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Semi-regular assignments</td>
<td>96</td>
</tr>
<tr>
<td>6.1.8</td>
<td>Inspector/Executor</td>
<td>97</td>
</tr>
<tr>
<td>6.1.9</td>
<td>Global-to-Local offset reuse</td>
<td>98</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>POST-SPMD OPTIMIZATIONS</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>RELATED WORK</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>HPF EXTENSIONS AND RESTRICTIONS</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>PARALLEL ASSIGNMENT</td>
<td>101</td>
</tr>
<tr>
<td>7.2</td>
<td>MAPPED SUBROUTINES OR FUNCTIONS</td>
<td>102</td>
</tr>
<tr>
<td>7.3</td>
<td>COMPOUND STATEMENTS</td>
<td>103</td>
</tr>
<tr>
<td>7.4</td>
<td>LOCAL VARIABLES INSIDE COMPOUND STATEMENTS</td>
<td>104</td>
</tr>
<tr>
<td>7.5</td>
<td>SCHEDULE EXTENSIONS</td>
<td>105</td>
</tr>
<tr>
<td>7.6</td>
<td>EXTENDED MAPPINGS</td>
<td>105</td>
</tr>
<tr>
<td>7.6.1</td>
<td>The identical distribution</td>
<td>106</td>
</tr>
<tr>
<td>7.6.2</td>
<td>HPF2 extended mappings</td>
<td>108</td>
</tr>
<tr>
<td>7.7</td>
<td>RESTRICTIONS TO THE HPF 2.0 LANGUAGE</td>
<td>112</td>
</tr>
<tr>
<td>7.8</td>
<td>RELATED WORK</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>EXPERIMENTAL RESULTS</td>
<td>115</td>
</tr>
<tr>
<td>8.1</td>
<td>THE TEST ENVIRONMENT</td>
<td>116</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Hardware: the Distributed ASCL Supercomputer</td>
<td>116</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Software: the HPF compilers</td>
<td>118</td>
</tr>
<tr>
<td>8.2</td>
<td>THE TNO-BENCHMARKS</td>
<td>118</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Initializing an array</td>
<td>121</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Non-Rectangular iteration spaces</td>
<td>125</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Copying an array</td>
<td>129</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Shifting an array</td>
<td>132</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Stencil assignments</td>
<td>134</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Sender Computes Rule</td>
<td>137</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Semi-regular assignments</td>
<td>138</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Conclusions</td>
<td>142</td>
</tr>
<tr>
<td>8.3</td>
<td>KERNEL TNO-Wish3d</td>
<td>142</td>
</tr>
<tr>
<td>8.4</td>
<td>COMPARING OVERHEAD OF DIFFERENT ENUMERATION/STORAGE METHODS</td>
<td>146</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Index Generation Overhead</td>
<td>146</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Storage Overhead</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>CONCLUSIONS</td>
<td>153</td>
</tr>
<tr>
<td>9.1</td>
<td>FUTURE RESEARCH</td>
<td>155</td>
</tr>
</tbody>
</table>

## APPENDIX

- COMMERCIAL HPF COMPILERS
- RESEARCH COMPILERS

## SUMMARY

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUMMARY</td>
<td>159</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAMENVATTING</td>
<td>161</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>173</td>
<td></td>
</tr>
</tbody>
</table>
List of examples

Example 1: Several HPF Processor directives. 16
Example 2: A Template directive. 16
Example 3: Several Distribution directives 17
Example 4: Several Alignment directives. 19
Example 5: Several On directives. 21
Example 6: Kinds of assignments 23
Example 7: Some Independent directives 23
Example 8: HPF data distribution and their normalized mappings 29
Example 9: Normalized volume assignments of Example 6. 30
Example 10: Examples of owners. 32
Example 11: Examples of basic assignments. 33
Example 12: Owner application: from mappings to owners. 34
Example 13: Mapping extraction: from owners to mappings. 35
Example 14: Sample HPF input 39
Example 15: The corresponding IR of the HPF sample of Example 14. 39
Example 16: Decomposition of a multi-owned assignment. 41
Example 17: Reduced communication by proper decomposition. 42
Example 18: Generalizing an owner. 42
Example 19: Lowering of an owner attributed scope. 43
Example 20: Usage of the global-to-local support function. 44
Example 21: SPMD code for a scalar assignment when $P_{\text{recv}} \subseteq P_{\text{send}}$ 58
Example 22: SPMD code for a scalar assignment when $P_{\text{recv}} \cap P_{\text{send}} \neq \emptyset$ 59
Example 23: SPMD code for a scalar assignment when $P_{\text{recv}} \cap P_{\text{send}} = \emptyset$ 60
Example 24: SPMD code for a volume assignment when $P_{\text{recv}(I)} \subseteq P_{\text{send}(I)}$ 61
Example 25: SPMD code for a volume assignment when $P_{\text{recv}(I)} \cap P_{\text{send}(I)} \neq \emptyset$ 62
Example 26: SPMD code for a volume assignment when $P_{\text{recv}(I)} \cap P_{\text{send}(I)} = \emptyset$ 63
Example 27: Communicate the size information 64
Example 28: Making the destination explicit when packing 65
Example 29: The most general one-dimensional HPF mapping. 70
Example 30: A naive owner implementation 74
Example 31: Mask absorption 75
Example 32: The broad applicability of mask absorption 76
Example 33: Hoisting out the global-to-local function. 79
Example 34: Merging messages 88
Example 35: Hoisting an assignment 89
Example 36: Mask absorption for independent loops 90
Example 37: A stencil optimization 91
Example 38: A reduction assignment 93
Example 39: A volume assignment with a mask 94
Example 40: Lowering of masked assignments 95
Example 41: Several masked assignments. .......................... 96
Example 42: Lowering of an irregular assignment. ................. 96
Example 43: Hoisting out expensive schedules .................. 97
Example 44: Scalar global-to-local offset reuse ................. 98
Example 45: Global to local offset reuse ....................... 98
Example 46: Parallel assignment ................................. 101
Example 47: Example of a parallel assignment .................. 102
Example 48: Mapped functions .................................. 103
Example 49: Usage of an independent compound statement in a Red-Black solver. 104
Example 50: Allocatables compared with local variables inside a compound statement 104
Example 51: Schedule extensions ................................ 105
Example 52: Usage of the index specifier @ ....................... 106
Example 53: Motivating example for the identical distribution. 107
Example 54: Distribution extensions ............................. 108
Example 55: Emulating an INDIRECT distribution ............... 109
Example 56: Example code using a two-dimensional mesh ...... 110
Example 57: Rewritten version of Example 56 using a one-dimensional mesh .................. 111
Example 58: A sample template (sample.tmpl) .................. 120
Example 59: Sample-S2-D3-I2-A4.hpf ............................. 121
Example 60: Init1D.tmpl .......................................... 122
Example 61: Init2D.tmpl .......................................... 126
Example 62: Copy1D.tmpl ........................................ 130
Example 63: Shift1D.tmpl ........................................ 132
Example 64: Stencil1D.tmpl ...................................... 134
Example 65: Stencil optimization (stencil offset <= 5). ....... 135
Example 66: No stencil optimization (stencil offset > 5). ....... 136
Example 67: Scr1D.tmpl .......................................... 137
Example 68: Semireg1D.tmpl .................................... 139
Example 69: The TNO-Wish3d Kernel code ....................... 144
Example 70: Test assignment for index generation overhead .... 147
List of figures

Figure 1: A shared-memory machine. 7
Figure 2: A distributed-memory machine 9
Figure 3: Data layout of the distributions as specified in Example 3. 18
Figure 4: Alignment examples. 20
Figure 5: Format of a normalized data mapping 27
Figure 6: Format of an owner 31
Figure 7: The CoSy framework. 37
Figure 8: PIR: CCMIR data parallel extensions 45
Figure 9: PERTS processor declaration support 46
Figure 10: PERTS mapping declaration support 47
Figure 11: PERTS mapped variables support 48
Figure 12: PERTS support functions 48
Figure 13: A common sub-expression: A + B 49
Figure 14: Owner reduction algorithm. 50
Figure 15: Deriving a parent owner by owner reduction. 51
Figure 16: Deriving owners by owner generalization and owner privatization. 52
Figure 17: Constant and variable owners of A(i) = A(i) + B(i). 53
Figure 18: A(i) = A(i) + B(i) after owner selection 55
Figure 19: Owners and masters of lhs and rhs expressions of the assignment x = y 56
Figure 20: The template data layout 70
Figure 21: No compression, Column-wise, Row-wise, and Tile-wise compression. 72
Figure 22: Intersection of local iterations of A(i, i, i) 80
Figure 23: A neighbouring stencil assignment 92
Figure 24: Buffer layout with mask piggybacking. 94
Figure 25: Layout of array A. 107
Figure 26: An irregular two-dimensional mesh. 110
Figure 27: A snapshot of the DAS cluster at the "Vrije Universiteit" 117
Figure 28: Model of theTNO-Wish3d application 143
Figure 29: Timings of Kernel TNO-Wish3d, for a (block, *, *) distributed template. 145
Figure 30: Execution times of simulated versions for a number of enumeration methods. 148
Figure 31: The execution time of the enumeration methods for a number of distributions. 149
Figure 32: Row-wise and Column-wise storage compression. 151
List of Tables

Table 1: Terminology and their domains
Table 2: Init1D: Initializing a one-dimensional array
Table 3: Init2D-A1: Diagonal iteration space.
Table 4: Init2D-A2: Triangular iteration space.
Table 5: Init2D-A3: Convex iteration space.
Table 6: Copy1D: Copying a 1-D array
Table 7: Shift1D-A1: A(i) = B(i+1)
Table 8: Shift1D-A2: A(i) = B(i+10).
Table 9: Stencil1D-A1: A(i) = B(i) + B(i+1).
Table 10: Stencil1D-A2: A(i) = B(i) + B(i+10).
Table 11: Sender Computes rule: A(i) = B(i+10) + D(i+10).
Table 12: Semireg-A1: Gather without communication
Table 13: Semireg-A1: Gather with communication
Table 14: Semireg-A3: Scatter without communication
Table 15: Semireg-A4: Scatter with communication
With the appearance in 1992 of High Performance Fortran (HPF) [32], a de-facto standard for data-parallel languages emerged. Together with the availability of cheap off the shelf computing power, like a cluster of workstations, parallel programming is now becoming widely used among scientists and industrial users.

The ease of programming in HPF gives it a great potential to easily migrate from existing sequential programs to a parallel version. The HPF language is designed such that programmers do not need to know the details of architecture of the parallel machine on which their application must run. In fact, a parallel HPF program might be a sequential program with properly added comments containing data distribution directives.

Such a program is still capable of running on a shared-memory machine, a distributed-memory machine, and even on a sequential machine. HPF is designed to make parallel programs portable over different parallel machines, with comparable performance. For the ultimate performance on a specific type of machine, fine-tuning is still needed.

Now HPF was defined, compilers still needed to be designed and implemented. That is what this thesis is about: making a clean design of an HPF compiler embedded in an existing F90 compiler. So how did this all start?

1.1 History of the research

Mid 1992 the European ESPRIT project EP 6516 PREPARE [73] started developing a Programming Environment for Parallel Architectures. The main contractor of PREPARE was the compiler manufacturer ACE (The Netherlands). Other industrial partners were Steria (France), SoftLab (Sweden) and Parsytec (Germany). Academic partners came from GMD-FIRST (Germany), IRISA (France), PELAB (Sweden), TNO-TPD (The Netherlands), the Technical University Munich (TUM) (Germany) and the University of Vienna. The total manpower for this project was 69 man-year; it began in July 1992 and ended in April 1996.

At the beginning of the project, there was no clear standard for a data parallel language. Some of the research languages available were Fortran D of Rice University [24], and Vienna Fortran [76]. Both languages were based on Fortran 77 and added their own attributes to it for work and data distributions. In addition, something similar was planned to be used in PREPARE.

However, it soon became clear that High Performance Fortran as developed by the HPF Forum would become a more widely accepted standard. HPF however was based on Fortran 90, with all its different types of arrays, array assignments, and array intrinsics. The shift from Fortran 77 to Fortran 90 greatly increased the complexity of the language as compared to Fortran 77.

---

1 This was in the beginning widely mistaken as all or most programs can be parallelized that way and the compiler will take care of the rest. This greatly hindered the success of HPF.
The objective of PREPARE was then shifted to build a programming environment in which High Performance Fortran (HPF) programs can be developed and restructured in a machine-independent fashion and executed on a massively parallel MIMD machine. The full Fortran 90 standard and the official subset of HPF needed to be supported by the PREPARE compiler. The PREPARE compiler was planned to be available by November 1995. A running beta-test version of the compiler was delivered at the end of the project.

The PREPARE environment was based on three tightly integrated components:

- A **Parallelization Engine** that transforms the original data-parallel form of the source program into a Single Program, Multiple Data (SPMD) form.

- An **Interactive Engine** that reports to the programmer to what extent the system can parallelize the program. It indicates the obstacles preventing parallelization, facilitates the removal by the programmer of such obstacles, and provides performance measures.

- An efficient **Compilation System** that generates highly optimized code that fully exploits the inter-processor parallelism of the target machine.

The compilation system used in PREPARE was based on the CoSy compilation system as developed in the sister project COMPARE [6]. The innovative compilation system makes it possible to configure highly optimizing compilers from a large set of building blocks called engines. These engines work concurrently on a shared Internal Representation of the program gradually enriching it with analysis information.

On a (shared-memory) parallel host, the engines can work in parallel making it possible to implement very computation intensive optimizations. Thanks to this modular structure, compilers based on the CoSy mechanism can be easily upgraded in order to support new parallel machines as well as new language standards.

In the PREPARE compiler all engines communicate with each other through the Internal Representation. The level of the Internal Representation has been carefully chosen to be appropriate for different kinds of engines. Those high-level engines that work with features that are close to the source program (such as the Parallelization Engine or the Interactive Engine), as well as for low-level engines (such as traditional optimizers or code generators).

In order to be able to communicate with the user in terms of the original source code an elaborate mechanism is implemented that maintains appropriate source level origin pointers even during extensive program transformations. This also proved to be very helpful in debugging the compiler itself.

In the FUTURE project [26] many of the lessons learned from the PREPARE project were used to redesign the compiler. One of these learned lessons was that a complete, robust, and fully validated Fortran 90 compiler was needed as a building block for further extensions. Because the redesign was so rigorous, the compiler was named the FUTURE HPF compiler.
With the redesign, the engines were structured such that the integration of data-parallel engines would interfere as little as possible with existing sequential engines. The different types of internal data-parallel specific attributes were minimized as far as possible.

By being a member of the HPF Forum we helped to define the HPF language and gave input to iron out the inconsistencies in the language specification of Version 1.0. For the second version, we mainly concentrated on the efficient implementation of different language features. That resulted in less but more efficient language features of Version 2.0.

We are still participating in the HPF forum by proposing new language features. Features, which have proven to be efficient, fit nicely into the data distribution model, and really needed by industrial applications. The required extensions are presented in Chapter 7.

1.2 Motivation of the research

Because HPF is a rather young language there are currently only a few compilers supporting the language. These compilers are mainly focused on getting the complete HPF language compiled. However to build a compiler qualified as a HIGH PERFORMANCE Fortran compiler the generated code needs to be very efficient with respect to speed and memory usage. There may be no efficiency loss, compared to a Fortran 77 compiler, on pure Fortran 77 programs. In addition, the efficiency of a data parallel HPF program must be comparable to an equivalent program, which uses a standard message-passing library, like PVM [27] or MPI [51].

This thesis deals with the efficient compilation of an HPF program with data distributions on a distributed-memory machine, because distributed-memory machines are envisioned to be the future low-cost parallel machines. It describes a mapping model that elegantly integrates data and work distributions within a sequential compiler framework. It shows where the HPF language can be extended to get a more expressive handling of mapped tasks and data within the same mapping model. The broad range of optimizations possible with the underlying mapping model are given by example.

Because the framework needs to be embedded in a sequential compiler, it faces other design criteria than what is usual in a research compiler where the context is molded according to the requirements of the topic of interest. For instance, the focus might be on the data alignment problem (where newly created variables need to be aligned with the existing ones), or the efficient communication of values among processors. In our prescribed framework, both of the previous mentioned research topics need to be integrated within it.

---

2 In mid 90's, the compilers were mainly focused on getting the complete Fortran 90 language compiled.
3 In the beginning, some compiler vendors only concentrated on the speed aspect and neglected the memory aspect. They allocated for distributed arrays the complete global array size on each processor, thereby rendering a large class of data intensive applications as unparallelizable.
4 Some loss of efficiency is accepted as long as it allows a simpler programming model.
1.3 The research question

The following question is posed in this thesis. Does there exist an HPF compiler design that:

- Allows processing of all allowed HPF distributions?
- Can handle both data and work distributions efficiently?
- Does support optimizations, in a flexible and orthogonal way?
- Can be implemented in a simple and robust way?
- Can be easily integrated within an existing sequential compiler?

We believe we have found such a design. Different chapters of this thesis will deal with each of the posed sub-questions.

1.4 An outline of this thesis

In Chapter 2, the compiler targets for which the compiler framework needs to generate code are described. Two target hardware architectures are presented with their typical characteristics that make parallelization difficult. The parallel abstract machine, which hides all parallel machine specific details, is briefly described. The Single Program, Multiple Data execution model is explained.

Chapter 3 introduces HPF. It describes the history of HPF and all its data distributions, work distributions, and different kinds of assignments. Each data distribution directive is given by example and it is shown how it affects the normalization of the data mapping. Similar examples are given for the different kinds of assignments that are allowed in HPF. It describes how different kinds of assignments are modeled into a single normalized volume assignment.

In Chapter 4, the compilation model is presented. The owner concept is introduced separately as an internal representation because it embodies the crucial parts of work distribution. The owner concept can also be used to specify distributions of iteration spaces. The close relation between a mapping (an internal normalized representation for data distributions) and an owner (an internal normalized representation for work distributions) is mentioned. The internal compiler framework is given, and the design criteria that need to be fulfilled. It shows how owner-aware engines interact with ‘classical’ engines. For each owner aware engine, it is shown how they produce or use the owner extensions. The alignment analysis as performed by the align-engine is discussed separately. The same holds for the lowering of basic assignments as performed by the SPMD engine.

Chapter 5 dives into the details of how to store and enumerate elements of distributed variables efficiently. Special attention is given to the storage compression technique to avoid wasting memory. The enumeration technique is designed in such a way that even compressed distributed variables can easily be referenced when enumerating them. A simple induction variable can be used to hold the addresses of the enumerated elements.
Pre- and Post-SPMD optimizations are given in Chapter 6. Again, by giving examples the potential of the optimizations within the framework are demonstrated without diving into details. Most of the optimizations can be used in different optimization schemes. Like one after the other, mixed, or applied repeatedly, all with a local effect, thereby guaranteeing an optimization sequence that best suits the given context.

The possible HPF extensions fitting within the same compilation model are presented in Chapter 7. Most of the extensions are representations already used internally that are made visible at the HPF source level. It will allow programmers to specify in detail how data and work should be distributed. It also allows implementing a compiler that does not do a great deal of analysis. Later when analysis is implemented in the compiler, the programmer does not need to specify so elaborately, while the compiler can still deduce most of the left out data and work directives.

A compiler based on the compilation model as described in Chapter 4 was not available for experiments, due to shortage of manpower. A prototype compiler based on a predecessor of the compilation model was used instead. The results of some experiments compared with a commercial HPF compiler are given in Chapter 8. The test environment is described including the TNO-Benchmark suite. From this suite, different kinds of communication patterns are evaluated separately. The timing results of these experiments helped us to find and tune the current compilation model.

In Chapter 9, the conclusions that can be drawn from our research are summarized. Some future research on the compilation model is outlined.

1.5 Contributions of this research

The main contribution of this research is that we have found a simple, symbolic, and intuitive internal representation that can be used to specify both data as well as work distributions. This representation can be used by the compilation system to reason about locality of data and placement of work. Furthermore, the representation is designed such that it fits elegantly into an existing sequential compiler and that new 'parallel' optimizations do not interfere with the more classical 'sequential' optimizations.

We also managed to combine the efficient closed form enumeration of local elements with their compact storage at the same time. Enumeration of element sets is generated as parameterized code per dimension of the iteration space. This results into great flexibility with respect to the shape of the iteration space. For instance, triangular iteration or more complex non-rectangular iteration spaces can be handled with the same compilation system. Other HPF compilers are mainly restricted to rectangular iteration spaces.

We also made a start with owner analysis, be deviating from the widely used "Owner Computes Rule", i.e. the owner of the left-hand side of an assignment is responsible for the evaluation of the complete assignment. Within our design, the compiler is free to select the best possible owners for each of the sub-expressions that occur within an assignment. The same technique can also be
used within basic blocks and even within a procedure body, although the analysis will become more complicated.

Finally, we have shown that considerable performance can be gained by enumerating *block_cyclic*-distributed elements in the proper order, i.e. column-wise or row-wise. It also avoids performance irregularities between slightly different distributions.
2 Compiler targets

A program written in HPF is designed to be executable on different kinds of architectures: uni-processors, symmetric multiprocessors (SMP), clusters, and networks of workstations (NOW). Whatever the architecture, the memory model roughly falls into two categories: shared-memory machines and distributed-memory machines.

Both memory models and their characteristics are discussed in Sections 2.1 and Section 2.2. The Parallel Abstract Machine (PAM) onto which the compiler is built and which hides the memory model details from the compiler is described in Section 2.3. The execution model the compiler will comply to is described in Section 2.4.

2.1 Shared-memory machines

In a shared-memory machine, all processors (processing elements or PE's) can access one shared memory as shown in Figure 1. The details of keeping the memory consistent, when multiple processors are reading and writing data to the shared memory, are hidden to the compilation system. A shared memory can be fully implemented in hardware, or fully in software, or a mixture of both.

![Shared-memory machine diagram](image)

Figure 1: A shared-memory machine.

For efficiency reasons, usually every processor has its own cache memory to feed the processors with data at a higher bandwidth. Complex cache coherency protocols are needed to keep data, pre-fetched into the different caches, coherent with the main memory and among each other.
Because all processors must be able to address the complete memory, a complex interconnection network is needed. Multiple simultaneous accesses at the same memory location are serialized. The processors must now contend with each other to gain access to a single resource, the memory. This memory contention is the main reason why shared-memory systems do not scale [58]. In practice, the number of processors is limited to 16 or 32 processors.

In a uniprocessor machine, the best performance for a single cache can be achieved when the cache has a high hit-rate (typically 95% or higher). In that case, the same memory reference occurs repeatedly and its value can be reused from the cache instead of the much slower main memory.

Shared-memory machines with multiple caches referencing memory locations already cached elsewhere will introduce inefficiencies, due to the memory coherency that has to be enforced. Therefore, the best performance can be achieved when data references hardly overlap. Data distributions as those allowed in HPF, when carefully chosen, might avoid messing up caches of other processors. Although we are considering shared-memory machines, from a programming point of view, we might view them as a distributed machine to get efficient cache behavior.

Because each processor can run its own code independently from the other processor careful synchronization of the different processes (or even threads) running on different processors is needed. Distributed processes (or threads) can be synchronized with a distributed variant of locks and semaphores. Communication between processors is handled by updating the shared memory. Reading or writing to the shared memory only involves one processor (one-sided communication).

From a language point of view there is little standardization in how to program a shared-memory machine in a machine independent way. An early attempt for standardization ANSI X3H5 [46] failed. Recently, with the introduction of OpenMP [52], which largely builds upon X3H5, a new attempt in standardizing shared-memory programming is made.

2.2 Distributed-memory machines

The main parallel machine, for which our compilation system will generate code, is a distributed-memory machine. Nowadays a simple network of workstations (NOWs) or PCs connected with a fast network is cheap, exists of common off the shelf components and is easily upgradeable and extendible. Moreover, as we will show later in Chapter 8, it performs for some applications very close to a specially engineered supercomputer like a Cray-T3E.

Code generation is however more complex for a distributed-memory machine than it is for a shared-memory machine. For a distributed-memory machine, the compiler does not only have to figure out the work distribution, it also has to consider the locality of data with respect to the

---

5 Although a programmer has no direct influence on the cache behavior, he/she can influence it indirectly.
processors\textsuperscript{6}. Code generation for a shared-memory machine can be modeled as a special case of code generation for a distributed-memory machine.

Our model of a distributed-memory machine contains a set of independent processing elements each with its own memory, its own operating system, and a network connecting all the processors (see Figure 2).

\begin{center}
\begin{tikzpicture}
  \node[draw, rectangle] (a) at (0,0) {Mem};
  \node[draw, rectangle] (b) at (1,0) {Mem};
  \node[draw, rectangle] (c) at (2,0) {Mem};
  \node[circle,fill] (d) at (0.5,-1) {};  
  \node[circle,fill] (e) at (1.5,-1) {};  
  \node[circle,fill] (f) at (2.5,-1) {};  
  \draw (a) -- (d);
  \draw (b) -- (e);
  \draw (c) -- (f);
  \draw (d) -- (e);
  \draw (e) -- (f);
\end{tikzpicture}
\end{center}

Figure 2: A distributed-memory machine

The topology of the network is not known to the compiler. A network topology might be a hypercube, a torus, or a shared bus. To the compiler, all processors can send or receive messages to or from all other processors. All data must reside in the local memory of one or more processors. These processors are called the owner of the data. When multiple processors own a copy of certain data, the data is said to be replicated over the processors. It is the responsibility of the compiler to keep the copies synchronized.

Distributed-memory machines communicate data among processors by sending and receiving messages. This implies that both the sending processor(s) and the receiving processor(s) are involved\textsuperscript{7} in the communication (so-called two-sided communication). This will restrict the possible interactions between the processors, and complicate the analysis and translation of the program.

Generally, two characteristics of a communication network dominate its performance: latency and communication bandwidth. Latency is defined as the elapsed time between the moment the

\textsuperscript{6} In a shared-memory system, it is profitable to have some form of knowledge about locality, but it is not strictly needed.

\textsuperscript{7} Either in (un)packing of the messages or synchronizing memory server threads.
first byte leaves the sending processor and the moment the first byte arrives at the receiving processor. The communication bandwidth between two processors is defined as the number of bytes that can be transported per second.

Latency can also be expressed as the number of bytes that can be sent before the first byte is received. For a cluster of workstations, latency can easily be in the order of 100 to 1000 bytes. In our distributed-memory machine model we therefore assume that for each distributed-memory machine it is always more efficient with respect to execution time to merge several small messages into one large message than to send them separately.

2.3 **Parallel Abstract Machine (PAM)**

Because the hardware of parallel machines can differ significantly, a run-time system is needed that hides these hardware differences from the compilation system. For that purpose, a *Parallel Abstract Machine* (PAM) has been designed in the PREPARE project. The PAM, a special software layer in the form of a library, has to be linked together with the object code. It contains compiler known functions.

Such a Parallel Abstract Machine is used to; hide the details of process synchronization and data transfer; facilitate the bookkeeping of distributed arrays; pack and unpack messages. It can be build on top of standard message passing libraries like PVM [27] or MPI [51]. The PAM models a **homogeneous system**; i.e. all processors are of equal type.

The PAM hides the details of shared- and distributed-memory models, by treating all of them as distributed-memory machines, which need two-sided communication. The communication on a shared-memory machine however can be implemented by just handing over a pointer to the receiving processor(s).

A specific part of the Parallel Abstract Machine is the **Parallel Engines Run-time System** (PERTS) that contains all distribution specific functionality like bookkeeping details of the distributed arrays.

The compilation system may generate calls to the compiler known functions of the PAM in the generated object code, and link in the PAM to make it an executable that can be executed on each processor.

Using a PAM improves the portability of HPF code among parallel machines. A drawback however is the fact that no machine specific optimizations are possible anymore, i.e. all generated code must adhere to the PAM function interface even when certain machines would prefer another interface. By designing the PAM as a very thin layer on top of machine optimized communication libraries, this drawback could be kept acceptable.

---

8 The CoSy compilation system allows code generation for a heterogeneous system, because it allows multiple back-ends generating code from the same Internal Representation. Together with MPI, which allows communication among heterogeneous systems, a heterogeneous PAM can be build.
After all, HPF is designed for good performance on a wide range of parallel machines, without getting the ultimate performance on a specific parallel machine. In other words, portability is expected to be of a higher priority to a user than the utmost performance.

2.4 Single Program Multiple Data execution model (SPMD)

The CoSy HPF compiler will generate code, which follows the Single Program, Multiple Data (SPMD) [37] execution model. It generates only one parameterized executable for all processors. This simplifies the code generation and the loading of the executable onto the different processors. The SPMD execution model allows generating code independently of the number of processors.

All processors will run their codes independently. The only places of synchronization are where data needs to be exchanged among processors. For shared-memory systems, this can simply be achieved by passing a pointer to the data. For distributed systems, explicit send and receive statements need to be inserted. This kind of synchronization is called loosely synchronous.

Besides its normal parameters, the SPMD executable is invoked on all processors with two additional parameters np and pid. The parameter np is the number of processors running the executable and pid is a processor identification number ranging from 0 to np-1. All processors will execute the same code but, depending on the processor identification number pid, they will execute different parts of the executable and refer to different parts of a distributed array.

While the SPMD program is running, the number of processors involved in the execution is constant. No extra processors can join in the execution of the SPMD program. Some parts of the SPMD program may be executed by a smaller set of processors. The processors not involved are temporarily masked out and waiting to join the execution again.
High Performance Fortran (HPF) is a language definition designed to simplify the task of programming data parallel applications. Originally, it was intended for distributed Multiple Instruction, Multiple Data (MIMD) machines. Recently, with the rise of different types of parallel architectures, like shared-memory symmetric multiprocessing (SMP) and distributed-shared memory (DSM), the programmability of these machines have become increasingly important. As such HPF reduces the programming complexity and increases the program portability by providing a single programming model for both kinds of machines.

Although HPF is not designed specifically for shared memory machines, data distributions still make sense on these machines. By specifying data distributions, the compiler will generate code such that each processor will mainly access a segment of the shared memory, without interfering with memory segments of other processors.

It is generally accepted that the largest obstacle to the widespread use of distributed-memory message passing systems is the difficulty encountered in programming them. It is necessary to explicitly partition data, organize message-passing instructions, and handle boundary cases, etc., which is a very complicated, time-consuming, and error-prone task. It also impairs the adaptability and portability of the resulting programs.

HPF removes this burden from the programmer. It comprises a set of extensions to the standard Fortran language. The central idea of HPF is to augment a standard Fortran program with directives that specify the distribution of data across memory, which is physically distributed but can be shared from the programming point of view (virtually shared).

The HPF compiler then handles the intricate problem of partitioning the data according to the data distribution directives and allocating computation to processors according to the locality of the data references involved. This includes inserting instructions necessary for data transfer depending on the target architecture, for example, by message passing or by a (possibly virtual) shared-memory mechanism.

HPF is also designed to be largely architecture independent. It can be implemented across the whole spectrum of multi-processor architectures: distributed- and shared-memory machines, MIMD and SIMD machines, vector machines, workstation networks, etc. It can even be implemented on single-processor systems because data distribution is specified by means of directives, i.e., structured comments, which do not affect program semantics.

These directives are significant only to an HPF compiler, but not to an “ordinary” Fortran compiler. The only extensions HPF made to the Fortran 90 language were the Forall statement and
For all construct and the PURE attribute\(^9\). HPF was designed to make the programming of distributed-memory systems easier without sacrificing the portability of the resulting programs.

### 3.1 History

The first version of the language definition, HPF 1.0, was finished in 1993 following an initial meeting convened in January 1992 by Ken Kennedy from Rice University and Geoffrey Fox from Syracuse University. The design of HPF's data distribution features was strongly influenced by FortranD and Fortran 90D (Fox et al. [24], Wu and Fox [75]) and Vienna Fortran (Zima et al. [76]).

There were several predecessors of HPF, which provided significant contributions to the language development. They allowed data distribution specifications by extending existing programming language like Fortran and C or newly defined languages. Examples of these are SUPERB (Bast et al [6]), Distributed Fortran (Merlin [49]), Pandore (Andre et al. [5]) and Booster (Paalvast et al. [54]).

The **For all** statement was first introduced in Kali (Koelbel [44]). HPF was also influenced by various dialects from vendors such as Digital (DEC [20]), Convex, Cray (Pase et al. [57]), IBM (Sanz et al. [64]), MasPar (MasPar [48]) and Thinking Machines Corporation (TMC [70]). Input into the HPF development also came from a variety of other sources (see HPF Forum [32] for further references).

Thinking Machine Corporation was quite successful with CM-Fortran for their Connection Machines (CM-2 and CM-5). The execution model of CM-Fortran was that the host executes the main program except for the array assignments. An array of (Single Instruction Multiple Data SIMD) processing elements would execute the array assignments. Quite a number of applications were written in CM-Fortran and could be considered as 'legacy code' for HPF. Because of the large CM-Fortran share in parallel Fortran applications, the porting of CM-Fortran applications to HPF was designed to be easy.

The HPF 1.0 language definition was published by the HPF Forum [32]. This document also defined an official subset of the language, called Subset HPF, to facilitate early implementation. In particular, Subset HPF did not contain the full Fortran 90 basis of HPF 1.0, but only Fortran 77 with a subset of Fortran 90 features. In practice, all early HPF implementations, starting to appear in 1994, concentrated on Subset HPF features rather than attempting to support all of HPF.

A number of features that were considered but not accepted into HPF 1.0 were presented in a separate document, the "HPF Journal of Development" (HPF Forum [32]). These features were rejected because of lack of time or consensus, or in order to minimize direct extensions to Fortran

---

\(^9\) When Fortran 95 appeared, these extensions were incorporated into the language. From that moment on, HPF consisted only of compiler directives on top of Fortran 95.
90, rather than because of technical flaws, and so were documented to be used in future language design activities.

A second set of meetings of the HPFF from April to October 1994 concentrated on corrections, clarifications, and interpretations of HPF 1.0. These activities resulted in HPF 1.1, a revised and corrected version of the language specification. Some requirements for future versions were also identified at this time.

A third set of meetings of the HPFF was held from January 1995 to February 1997 to develop further extensions to HPF. The aim of these was to broaden HPF's applicability by providing features such as enhanced data distribution, task parallelism and computation control, parallel I/O, and directives to assist communication optimization. However, it became clear that vendors were reluctant to greatly extend the basic HPF language out of fear for delaying commercial implementations and/or encouraging partial implementations, thus undermining HPF's credibility and use.

The outcome was the definition of a new HPF base standard, HPF 2.0, which is quite similar to Subset HPF in terms of its HPF extensions. Probably the major difference between Subset HPF and HPF 2.0 is that the former was based on Fortran 77 and a subset of Fortran 90 features (e.g. array syntax), while HPF 2.0 is based on full Fortran 95. All other HPF extensions not part of HPF 2.0, both new and old, are designated HPF 2.0 Approved Extensions.

The idea was that a standard conforming HPF 2.0 compiler would provide full support for the HPF 2.0 features, but should not be required to support any of the Approved Extensions. Presumably the Approved Extensions will tend to be provided only if there is sufficient demand from users. The status “Approved Extension” should make it possible to be more flexible to deviate from the detailed specification, possibly allowing improvements in the light of implementation and user experience. Those features that turn out to be widely used can then be incorporated into the next version of the HPF base standard.

### 3.2 Data distribution directives

Because data mapping is the starting point of work distribution in a data-parallel language, we will discuss the different data mappings of HPF in more detail. HPF has a rich set of directives to specify data mappings. Each directive will be explained in the following subsections. Because HPF is an extension to Fortran 90 some of the terminology of Fortran 90 will be used where needed and are printed in bold. For details about Fortran 90, see [23].

#### 3.2.1 The PROCESSORS directive

The PROCESSORS directive declares one or more rectilinear processor arrangements, specifying for each one its name, its rank (number of dimensions), and the extent (number of ele-
ments) in each dimension. Every dimension of a processor arrangement must have a non-zero extent; therefore, a processor array can not be empty.

Processor arrangements are local entities of class (1) (see Section 14.1.2 of the Fortran 90 language standard); therefore a processor arrangement may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Names of processor arrangements obey the same rules for host and use association as other names in the long list in Section 12.1.2.2.1 of the Fortran 90 standard. Processor arrangements can not be passed as parameters.

If two processor arrangements have the same shape, then corresponding elements of two processor arrangements are understood to refer to the same abstract processor. The intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE may be used to inquire about the total number of actual physical processors. If no shape is specified, then the declared processor arrangement is conceptually scalar. Some processor arrangements are given in Example 1.

```
!HPFS$ PROCESSORS P(N)
!HPFS$ PROCESSORS Q(NUMBER_OF_PROCESSORS()), &
!HPFS$ R(8,NUMBER_OF_PROCESSORS())/8
!HPFS$ PROCESSORS BIZARRO(1972:1997,-20:17)
!HPFS$ PROCESSORS SCALARPROC
```

Example 1: Several HPF Processor directives.

An HPF compiler is required to accept any PROCESSORS declaration in which the product of the extents of each declared processor array is equal to the number of physical processors. It must also accept all declarations of scalar PROCESSOR arrays.

### 3.2.2 The Template directive

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank, and the extent in each dimension. A template is simply an abstract space of indexed positions; it can be considered as an “array of nothings” (as compared to an “array of integers”, say). A template may be used as an abstract align-target that may then be distributed.

Templates are useful in the particular situation where one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest. An example of how to use a template is given in Example 2:

```
!HPFS$ TEMPLATE :: EARTH(N+1,N+1)
!HPFS$ ALIGN NW(I,J) WITH EARTH(I,J)
!HPFS$ ALIGN NE(I,J) WITH EARTH(I,J+1)
!HPFS$ ALIGN SW(I,J) WITH EARTH(I+1,J)
!HPFS$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

Example 2: A Template directive.
Templates may also be useful in making assertions about the mapping of dummy arguments. Two templates, declared in different scoping units, will always be distinct, even if they are given the same name. The only way for two program units to refer to the same template is to declare the template in a module that is then used by the two program units.

Templates are not passed through the subprogram argument interface. The template to which a dummy argument is aligned is always distinct from the template to which the actual argument is aligned, though it may be a copy.

3.2.3 The Distribute directive

The `DISTRIBUTE` directive specifies a distribution of data arrays to abstract processors in a processor arrangement. The `REDISTRIBUTE` directive is similar to the `DISTRIBUTE` directive but is considered an executable statement. For each dimension of the distributee the distribution format can either be a `BLOCK(M)`, `CYCLIC(M)`, or collapsed (denoted with a `*`).

A `BLOCK(M)` distribution divides the extent of the distributee into blocks of size `M` and associates each block with a processor element. The blocksize `M` has to be positive and the number of blocks may never exceed the extent of the corresponding dimension of the processor arrangement. This imposes a minimum value of $M_{\text{min}} = \lceil N / P \rceil$, where $N$ is the number of elements in the distributed dimension and $P$ is the number of elements in the corresponding processor dimension.

A `CYCLIC(M)` distribution is the same as a `BLOCK(M)` distribution except that there is no restriction to the minimum value of $M$. When the blocksize is less than $M_{\text{min}}$ there are more blocks than processor elements, and each block is associated with a processor element in a round-robin fashion.

Two special distributions can be specified by omitting the size `M`. For instance, a `CYCLIC` distribution equals to a `CYCLIC(1)` distribution and a `BLOCK` distribution equals a `BLOCK(Max)` distribution. Where `Max` is the maximum block size such that the array’s extent is distributed evenly over the processors extent.

A collapsed dimension denotes that there is no relation between the data arrays dimension and the processor arrangement. In Example 3 several HPF distributions are given.

```hpf
TEMPLATE, DIMENSION(20) :: T1, T2, T3
TEMPLATE, DIMENSION(50) :: T4
!HPFS PROCESSORS P(4)
!HPFS DISTRIBUTE T1(BLOCK) ONTO P
!HPFS DISTRIBUTE T2(BLOCK(6)) ONTO P
!HPFS DISTRIBUTE T3(CYCLIC) ONTO P
!HPFS DISTRIBUTE T4(CYCLIC(4)) ONTO P
```

Example 3: Several Distribution directives

10 A `CYCLIC(M)` distribution with $M$ larger than one is also known in the literature as a `block_cyclic` distribution.
The memory layouts onto the processor arrangement of T1, T2, T3, and T4 are depicted in Figure 3. The memory layout of a CYCLIC(M) distribution (T4) results in a three-dimensional memory layout. The global memory position can be characterized by a (r, p, c) tuple, where p is a processor number, r a local row index, and c a local column index. A CYCLIC distribution (T3) is a special case of a CYCLIC(M) distribution and can be described by a (r, p, 0) tuple. Likewise for a BLOCK(M) distribution (T1, T2), which can be described with a (0, p, c) tuple.

To avoid confusion between HPF distributions types and internal compiler distributions types based on their tuple characteristics, we will from now on only use block for BLOCK distributions, cyclic for CYCLIC(1) distributions, and block_cyclic for CYCLIC(M) distributions with M larger than one, as the internal compiler distribution types.

Figure 3: Data layout of the distributions as specified in Example 3.
3.2.4 The Align directive

The ALIGN directive is used to specify that certain data arrays are to be mapped in the same way as certain other data arrays. Operations between aligned data arrays are likely to be more efficient than operations between data arrays that are not known to be aligned\textsuperscript{11}. The ALIGN directive is designed to make it particularly easy to specify explicit mappings for all the elements of an array at once. A REALIGN directive is similar to the ALIGN directive but it is considered an executable statement.

Example 4: Several Alignment directives.

The element associations of X, Y and Z with their templates are depicted in Figure 4.

\textsuperscript{11} A restricted set of alignments can also be specified by careful selection of distributions. Our compiler framework can detect both kind of alignments, irrespective of how they are declared.
With the ALIGN directive a functional (Z), a one-to-many (Y), or a many-to-one (X) relation can be specified between an array and a template. The concept of alignment facilitates the reasoning about the owner of an array element. An owner of an array element is the corresponding processor element or set of processor elements on which the array element resides. For instance, the owner of $X(i,*)$ is equal to the owner of $T5(i)$. The compiler can reason about equality of mappings when they both have the same alignment to the same template, without having to know the template distribution.

### 3.2.5 The INHERIT directive

The INHERIT directive specifies that a formal (a dummy in FORTRAN terminology) argument should be aligned to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned. If a dummy argument has the INHERIT attribute then the net effect is to tell the compiler to leave the data exactly where it is.

---

12 The * denotes that the value of the index is not needed to calculate the owner of the element.
3.3 Work distribution directives

3.3.1 The On directive

The purpose of the ON directive is to allow the programmer to control the distribution of work among the processors of a parallel machine. In a sense, this is the computation analog of the data distribution directives. The ON directive does this by specifying the active processor set and gives the programmer more control. If the computations in two ON blocks are not related (for example, if the ON blocks are two iterations of an INDEPENDENT loop), then they give the compiler clear instructions for exploiting parallelism.

The ON directive restricts the active processor set for a computation to those named in the directive. That is, it advises the compiler to use the named processor(s) to perform the computation. As the mapping directives ALIGN and DISTRIBUTE, this is advice rather than an absolute commandment; the compiler may override an ON directive.

Also like ALIGN and DISTRIBUTE, the ON directive may affect the efficiency of computation, but not the results. The values of the expressions used in the ON directive are determined as if they were evaluated when control flow reached the ON directive. Examples of ON directives are given in Example 5.

```fortran
!HPF$ INDEPENDENT
DO I = 1, N-1
   !HPF$ ON P(I)
   A(I) = (B(I) + B(I-1) + B(I+1))/3
   !HPF$ ON HOME(C(I+1)) BEGIN
     C(I+1) = A(I) * D(I-1)
   END
   C(I-1) = A(I) * D(I-1)
!HPF$ END ON
END DO
```

Example 5: Several On directives.

The location specified with the ON directive can be either a processor location or an ON HOME clause. The ON HOME clause specifies a processor location indirectly in terms of variable or template elements. An ON directive can attribute either a single executable statement or an executable statement block. Blocks with ON directives may be nested. It is legal to nest ON directives, if the set of active processors named by the inner ON directive is included in the set of active processors from the outer directive.

---

13 Block is a Fortran syntax term for "a series of statements treated as a group" (e.g. the body of a loop).
3.4 Kinds of assignments

In an assignment statement a right hand side (rhs) expression is evaluated and assigned to a location as given in the left hand side (lhs) expression. However, HPF knows many different kinds of assignments. Most of them are inherited from the Fortran 90 language on which HPF is based. HPF adds only one new assignment, the FORALL statement\(^\text{14}\).

Of all assignments, the scalar assignment is the simplest one. A scalar assignment is an assignment where only a single element is assigned.

An array assignment is the same as a scalar assignment except that the compiler can derive from the shape of the lhs expression a rectangular iteration space. One of the most important characteristics of array assignments is that for each rhs (sub)-expression, an array type can be derived for temporary variables to store an intermediate result. An extension to assignments containing mapped arrays will be given in Section 4.4.4.

The WHERE statement is the same as an array assignment except that it filters the iteration space with a mask expression. The elsewhere part is a subsequent WHERE statement with a negated mask\(^\text{15}\). The shape of the rhs and mask expressions must be conformable (See FORTRAN 90 standard [23]) to the lhs expression.

The FORALL statement is a more general assignment than the WHERE statement. It makes the loop variables scanning the iteration space explicit. For each point in the iteration space, the lhs expression must evaluate to a different array element. The rhs can be any expression with the same type as the lhs expression (just as in a scalar assignment). This allows rhs and mask expressions, which are non-conformable to the lhs-expression. Scanning diagonals or transposing arrays can be easily expressed\(^\text{16}\). The iteration space must be rectangular and may be followed by only one mask expression.

The FORALL construct allows multiple (array) assignments, WHERE statements, and nested FORALL constructs and statements to be controlled by a single iteration space specification. By nesting two FORALL constructs, non-rectangular iteration spaces can be scanned like triangular iteration spaces.

A volume-assignment is an assignment where multiple elements are assigned at once. The order in which the assignments are executed is not specified. The array assignment, the WHERE statement, the FORALL statement, and the FORALL construct are all volume-assignments. When evaluating a volume-assignment a fetch before store semantics holds. That means all values of

\(^{14}\) The FORALL construct is considered as 'syntactic sugar', because it is actually a sequence of FORALL statements.

\(^{15}\) This is not what most programmers will usually expect! Namely, two cases of assignments executed depending on the mask for a single sweep over the iteration space.

\(^{16}\) An attempt is made to allow more expressive array assignments by introducing special array intrinsic functions like SPREAD, which add additional dimensions to an array expression to make it conformable with the array on the lhs.
the rhs expression must be evaluated for the complete iteration space before they can be assigned to the lhs data elements.

Several kinds of assignments are given in Example 6.

```fortran
REAL Sc,A(10),B(10,10)

Sc = 1               ! Scalar assignment
A = 1               ! Array assignment
WHERE (A /= 0)       ! Where statement
  A = 1/A
ELSEWHERE
  A = 2*A
END WHERE
FORALL (I = 2:10) A(I) = A(I-1)    ! Forall statement
FORALL (I = 1:10)    ! Forall construct
  A(I) = I
FORALL (J = I:10)
  B(I,J) = A(I)*J
END FORALL
END FORALL
```

Example 6: Kinds of assignments

### 3.4.1 The Independent directive

The `INDEPENDENT` directive can precede an indexed `DO` loop or `FORALL` statement (see Example 7). It asserts to the compiler that the iterations in the following `DO` loop or the operations in the following `FORALL` may be executed independently (i.e. in any order, or interleaved, or concurrently) without changing the semantics of the program.

```fortran
!HPFS INDEPENDENT
DO I = 2,N
  A(I) = B(I-1) + B(I) + B(I+1)
END DO

!HPFS INDEPENDENT
FORALL (I = 2:N)
  A(I) = B(I-1) + B(I) + B(I+1)
END FORALL
```

Example 7: Some Independent directives

Strictly speaking, the `INDEPENDENT` directive is not a work distribution directive. It does however relax the control flow of the program, and thereby influences how work is scheduled. The
independent directive is actually analysis information that is provided by the programmer. The compiler is free to ignore the directive, or to complain if it can prove it is incorrect.
4 The Compilation Model

4.1 Terminology

To make a clear distinction between HPF language elements and compiler design elements a
different terminology is used in both domains. Some of these terms are related as is shown in
Table 1.

<table>
<thead>
<tr>
<th>Language terminology:</th>
<th>Compiler terminology:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directive</td>
<td>Attribute</td>
</tr>
<tr>
<td>Data directive</td>
<td>Mapping attribute</td>
</tr>
<tr>
<td>Work directive</td>
<td>Owner attribute</td>
</tr>
<tr>
<td>ON directive</td>
<td></td>
</tr>
<tr>
<td>BLOCK(M)</td>
<td>block</td>
</tr>
<tr>
<td>CYCLIC(1)</td>
<td>cyclic</td>
</tr>
<tr>
<td>CYCLIC(M)</td>
<td>block_cycle</td>
</tr>
<tr>
<td>Array assignment, WHERE,</td>
<td>Normalized volume assignment (forall)</td>
</tr>
<tr>
<td>FORALL</td>
<td>* Scalar assignment</td>
</tr>
<tr>
<td>! Comments</td>
<td>// Comments</td>
</tr>
</tbody>
</table>

Table 1: Terminology and their domains

A **directive** is a language element that has a form of a comment with an HPF prefix ‘!HPFS’. It
gives the compiler additional information about the language structure to which it applies. The
compiler counterpart is an **attribute**, which attributes the node in the internal representation (IR)
corresponding to the language structure.

There are two kinds of directives: **data directives** and **work directives**. Data directives specify
the distribution of data that needs to be applied when data is allocated. It is a type specification.
The compiler equivalent is a **mapping attribute**. Work directives specify where (on which pro-
cessor) work needs to be executed. Currently, in the language domain there is only one work di-
rective: the ON directive. Its compiler equivalent is an **owner attribute**. The owner attribute can
however attribute much more nodes in the internal representation than would be possible with
ON directives.
In the language domain there are only two distribution types: BLOCK or CYCLIC. In the compiler domain we distinguish however three kinds of distribution types: block, cyclic and block_cyclic. The correspondence between these types is given in Table 1.

All kinds of assignments in the language domain correspond to a normalized volume assignment in the compiler domain (denoted as a forall).

4.2 Normalization

The compilation system will normalize the parsed HPF program to keep the complexity of the following compilation phases as low as possible. Besides the standard normalization used for sequential programs, the compilation system specifically normalizes data mappings and assignments.

4.2.1 Normalizing HPF data mappings

All possible HPF data mappings can be represented in the compilation system in a single internal representation. Therefore, all possible HPF data distributions and alignments are normalized into a parameterized two-level mapping as depicted in Figure 5. A mapping is relation that maps a parameter list onto a processor location.

Arrays and scalars types can be attributed with a mapping. An array type that is attributed with a mapping specifies which array element is mapped onto which processor element. The number of parameters is equal to the rank of the mapped array. The first parameter binds to the first dimension of the array, and so on. A scalar type that is attributed with a mapping specifies the location where the scalar of that type resides. A scalar can be viewed as a zero-rank array and has an empty parameter list.

Each scalar or array mapping can be described by a composition of a distribution function $f_{\text{distr}}$ and an alignment function $f_{\text{align}}$, forming the two levels of the mapping. The distribution function can either be a replication (denoted with a *), constant, block, cyclic, or block_cyclic distribution function. The type of distribution function that is used in a dimension of the processor array is called the distribution type of that dimension. The alignment function is always a linear function. The base of the processor array, $P_{\text{expr}}$, can be any expression that evaluates to a processor array.
A: n-dimensional array

A: mapped \((i_0, \ldots, i_{n-1})\) on \(P_{\text{expr}}(f_{\text{distr}_0}(i_{k_0}), \ldots, f_{\text{distr}_n}(i_{k_{n-1}}))\)

\[0 \leq k_i \leq n - 1 \quad k_i \neq k_j\]

\[
\begin{cases}
  * \\
  \text{const} \\
  \text{block}(f_{\text{align}}(i), m_0) = i / m_0 \\
  \text{cyclic}(f_{\text{align}}(i), m_i) = i \mod m_i \\
  \text{block}_\text{cyclic}(f_{\text{align}}(i), m_0, m_i) = (i / m_0) \mod m_i
\end{cases}
\]

\(f_{\text{align}}(i) = a \ast i + b\)

\(a = \text{alignment stride}\)
\(b = \text{alignment offset}\)
\(m_0 = \text{block size}\)
\(m_i = \text{cycle length}\)

Figure 5: Format of a normalized data mapping

The binding with array indices is indicated with the formal parameters \((i_0, \ldots, i_{n-1})\). The alignment function may use only one of the formal parameters to bind an array dimension with a processor dimension. Not every formal parameter needs to be used. Unused formal parameters are called **collapsed dimensions** of the attributed array.

**Replicated dimensions** are represented with a * distribution function. Mappings containing one or more * distribution functions will map a single element onto a set of processors. The * can be seen as a wildcard specification of the set. A scalar mapping has the same mapping format but it has an empty list of formal parameters.

Notice that there is no representation for templates, and the compiler will not reason about mappings or owners in terms of templates. A nice feature of this representation is that mappings can be compared symbolically at the processor index level. For that reason the values of the alignment stride \(a\), the alignment offset \(b\), the block size \(m_0\) or the cycle length \(m_i\) does not need to be known at compile-time. The same holds for the shape of the array, its template, or the processor array.
HPF mapped arrays can either be directly mapped (an HPF distributed array), or indirectly mapped (an HPF aligned array). For an HPF distributed array (declared with ‘HPFS DISTRIBUTE’), a trivial alignment \( (a = 1, b = 0) \) is created for each distributed dimension. Distribution functions are created according to the distribute directive.

An HPF aligned array (declared with ‘!HPFS ALIGN’) will be attributed with a copy of the processor array base expression \( P_{expr} \) and the distribution functions. A new alignment function is constructed as a composition of its target alignment function and the alignment function as specified in the ALIGN directive.

The mapping specification may depend on values only known at run-time and therefore it can be viewed as a dynamic type attribute of these arrays. The run-time system therefore keeps all information about the data mapping into a run-time mapping descriptor\(^{17}\). Several arrays can share the same run-time mapping descriptor.

HPF transcription mappings (partially described mappings) and inherited mappings of dummy variables will copy (parts of) the run-time mapping descriptor of the corresponding actual argument into their own run-time mapping descriptor. The exact details of what to store inside a run-time mapping descriptor will be explained in Section 4.4.6.

For example, if array element \( A(i, j, k) \) is aligned with \( T(4*j, 8*i, *) \). And \( T \) is distributed (block, *, block_cyclic) onto \( P \), then array element \( A(i, j, k) \) is mapped simply onto \( P(block(4*j), *) \). This is denoted as ‘A \textit{mapped} (i, j, k) on P(block(4*j), *)’.

Note that the mapping specifies how an array element corresponds to a processor element. It does not specify how multiple elements owned by the same processor (the local elements of the processor) are stored locally\(^{18}\).

Once the mappings are normalized, several important characteristics are known at compile-time:

1. For all HPF mapped scalars and arrays, proper alignment and distribution functions can always be constructed.

2. For each array dimension its distribution type (*, const, block, cyclic or block_cyclic) is known at compile-time. In case of an unspecified distribution of an array dimension, a block_cyclic distribution can be assumed because all other distribution types are special cases of a block_cyclic distribution.

3. For explicit data distributions, the processor array can be traced from the specified alignment and distribution of the array. This processor array will be used in the normalized mapping. The rank of the processor array, the binding of array dimensions with processor dimension, and the distribution type is known at compile-time.

\(^{17}\) The mapping descriptor is a run-time object because parts of the information describing the mapping may depend on data known at run-time. Run-time type descriptors are also known as dope vectors.

\(^{18}\) This degree of freedom will be exploited when we use compressed storage (see Subsection 5.1.1)
4. For **implicit data distributions**, the processor array can not be traced. The compiler will assume a processor array with the same shape as the array to which it is ultimately aligned. A one-to-one binding is assumed between each dimension of the array and its associated processor array. Finally, the most general block cyclic distribution is assumed for each processor dimension\(^{19}\). All other information that is not known at compile-time is expressed as expressions reading from the array’s run-time descriptor.

In Example 8 the normalized mapping is given for several HPF data distributions.

```hpfs
REAL C(:,:,:)  
!HPFS$ DISTRIBUTE A(BLOCK, CYCLIC,*) ON TO P  
!HPFS$ ALIGN B(I,J,K) WITH A(3*I,J,*,I+4)  
!HPFS$ INHERIT C  
// internally attributed as  
A: mapped (i,j,k) on P(block(i),cyclic(j))  
B: mapped (i,j,k) on P(block(3*j),*)  
C: mapped (i,j,k) on C.P(block_cyclic(C.a[0]*i+C.b[0],C.m[0],C.m[0]),  
    block_cyclic(C.a[1]*j+C.b[1],C.m[0],C.m[0]),  
    block_cyclic(C.a[2]*k+C.b[2],C.m[0],C.m[0]))
```

Example 8: HPF data distribution and their normalized mappings

Because \(A\) and \(B\) are aligned to each other they share the same processor array \(P\). The processor array of array \(C\) is not explicitly known at compile-time. Therefore, the one that is stored in the run-time descriptor of \(C\) is used (i.e. \(C.P\)). The mappings of \(A\) and \(B\) are explicitly given, and the one of \(C\) is assumed. The alignment stride \(a\), alignment offset \(b\), block size \(m_0\), and cycle length \(m\) are read from the run-time descriptor of \(C\). See Section 4.4.6 for more details about run-time descriptors.

### 4.2.2 Normalized volume assignment

To simplify the compiler framework each (scalar or volume) assignment is normalized into a **normalized volume assignment**. A normalized volume assignment is an assignment with an iteration space that has no fetch before store semantics (i.e. it is equivalent to a FORALL with a ‘!HPFS$ independent’ directive). The normalized volume assignment is attributed such that the original kind of assignment leading to this normalized volume assignment is known. A scalar assignment will result in a normalized volume assignment with an empty iteration space. A normalized volume assignment always has a rectangular iteration space. The iteration space itself is normalized such that each dimension of the iteration space has a loop count and a list of induction variables each having its own lowerbound and stride.

---

\(^{19}\) The compiler is allowed to assume this, because the run-time descriptor can be initialized such that the assumptions are always true.
For volume-assignments, which can not directly be translated into a do-loopnest\textsuperscript{20} because loop-carryed dependencies\textsuperscript{21} might arise, the volume-assignment is split into a sequence of two volume-assignments. The first volume-assignment evaluates the rhs expression and stores the values for the complete iteration space into the temporary array. The second volume-assignment copies the temporary array into the lhs array.

Because of this splitting, normalized volume assignment can always be lowered\textsuperscript{22} straightforwardly into a do-loopnest with a scalar assignment in it. In the sequel of this thesis, we will talk about assignments when we mean normalized volume assignments. In Example 6 the normalized volume assignments of Example 6 are given. The variables starting with an underscore are created by the compiler. A normalized volume assignment is denoted with ‘forall’ (lowercase and bold printed) in contrast to the HPF language ‘FORALL’ (uppercase).

```
real Sc,A(10),B(10,10)
!	Iteration space mask	lhs = rhs
!	(cnt | var = wb:str)
forall ()	Sc1 = 1
forall (10 | _t = 1:1) A(_t) = 1
forall (10 | _t = 1:1) _MASK(_t) = (A(_t)!* 0)
forall (10 | _t = 1:1, _MASK(_t)) A(_t) = 1/A(_t)
forall (10 | _t = 1:1, _MASK(_t)) A(_t) = 2*A(_t)
forall (10 | I = 1:1)  
 forall (10 | I = 1:1) tmp(I) = A(I-1) ! fetch before store
forall (10 | I = 1:1) A(I) = tmp(I) ! removed
forall (10 | I = 1:1) A(I) = I
do (10 | I = 0:9) <independent>
forall (10-I | J = I:1) B(I,J) = A(I)*J
end do
```


The above normalized iteration spaces of volume assignments in following examples will not be used. The original iteration space notation will be used to keep the examples as simple as possible. The same holds for the extra parameters, \( m_0 \) and \( m_1 \) of the distribution functions. Whenever possible they will be omitted.

### 4.3 The Owner Concept

Before we can describe how the compilation framework is structured and how each engine (see Section 4.4) operates on the intermediate representation, we need to introduce the owner concept.

\textsuperscript{20} A do loop with FORTRAN semantics, i.e. all bounds are calculated before the loop is entered and a simple loop counter can be used to count down the number of iterations.

\textsuperscript{21} A loop-carried dependency occurs when a data element is read in one iteration and written in another (i.e. it carries its dependency over the loop).

\textsuperscript{22} Lowering is the translation of a high-level structure into a lower-level structure, and usually information is lost.
The owner concept and its characteristics are crucial for the comprehension of the different pre-SPMD optimizations (see Section 6.1).

An owner describes the location or locations where work should be executed. An owner is an expression that evaluates to an index in a processor array. The format of an owner is given in Figure 6. It resembles the format of a mapping (see Figure 5), except that there are no formal parameters and there is no separate alignment function. The actual argument of the distribution function in each dimension \( k \) of the owner can be an arbitrary integer valued expression and is denoted with \( expr_k \).

\[
\begin{align*}
\text{on } P_{expr} & (f_{\text{distr}_0}(expr_0), \ldots, f_{\text{distr}_{n-1}}(expr_{n-1})) \\
& \begin{cases}
  \text{#} \\
  \text{i} \\
  \text{block}(i,m_0) = i / m_0 \\
  \text{cyclic}(i,m_1) = i \mod m_1 \\
  \text{block_cyclic}(i,m_0,m_1) = (i / m_0) \mod m_1
\end{cases}
\end{align*}
\]

\[
m_0 = \text{block size} \\
m_1 = \text{cycle length}
\]

Figure 6: Format of an owner

The compilation system tries to determine the type of the distribution function, by matching the expression used in a dimension of an owner with the patterns of the distribution functions. The order of matching is first \( \text{block_cyclic} \), then \( \text{cyclic} \), then \( \text{block} \) or else the identity function.

With this distinction between distribution functions, we can easily express in our examples that the expression in a dimension of the owner is of a certain format, and is as such recognized by the compilation system.

An owner can be an attribute of an executable IR node: an expression, a statement, a basic block\(^{23}\), and even a procedure body. The scope of an owner is the scope of the internal representation (IR) node it attributes. The format of the owner is designed in such a way that it can be easily manipulated by the compiler. For instance, owners can be compared symbolically, ab-

\(^{23}\) A basic block is a sequence of statements that act as an atomic unit in the control flow.
sorbed into loops, hoisted in the internal representation hierarchy, and merged. Examples of these manipulations will be given as optimizations in Chapter 6.

4.3.1 Work distribution

An owner attribute of an expression, statement, a basic block, or a procedure body specifies the active processor set (See [33] for its definition) that is involved in the execution of the IR node (denoted with on <owner-list> {...}). Single-owned scopes have only one owner in the owner list. Two-owned scopes have exactly two owners. Multi-owned scopes have at least three owners. Some owners with their scopes are given in Example 10. In the first example only the processor set \( P(i) \) will be involved in the execution of the expression\(^{24} \). The second example guarantees that only processor \( P(i) \) is involved in the execution of the assignment. Both right-hand side and left-hand side expressions can be evaluated at \( P(i) \). The last example shows that only two processors \( P(i) \) and \( P(j) \) are involved in the execution of the basic block. For a part of the basic block, i.e. the expression \( B(j) \), only one processor \( P(j) \) is involved.

```
A,B: mapped(i) on P(i)

on P(i){ 2*A(i) }                     // single-owned expression

on P(i) {
    A(i) = B(i)
}

on P(i),P(j){
    A(i) = B(i)
    A(i) = on P(j){ B(j) }
}
```

Example 10: Examples of owners.

The owners in the above example can be found by mapping application see Section 4.3.3.

Multi-owned scopes can always be lowered (transformed) into a sequence of single-owned or two-owned scopes. Special alignment analysis and decompose transformations as described in Sections 4.4.2 and 4.4.3 are needed for this kind of lowering.

Notice that the owner attribute has a finer granularity than the ON directive. The ON directive has at least a statement scope, while the owner attribute can have an expression scope. This difference will become significant when the alignment engine (Section 4.4.2) and decompose engine (Section 4.4.3) are discussed.

\(^{24}\) Whether \( P(i) \) denotes a single processor or a set of processors may not be known to the compiler.
4.3.2 Iteration Space distribution

The iteration space of a normalized volume assignment (see Section 3.4.1) can also be attributed with an owner and specifies a distribution of the iteration space. In case the iteration space has only one (single-owned assignment) or two (two-owned assignment) owners, it is called a basic assignment (examples are given in Example 11). For multi-owned assignments special alignment analysis and decompose transformations must take place as described in Sections 4.4.2 and 4.4.3 resulting in the above single-owned or two-owned assignments.

For two-owned assignments, the order of the two iteration-space owner attributes is significant. The first owner in the list can be associated with the receiving owner, the owner where the actual assignment takes place. The second owner is the sending owner, which must provide the values of the (partially) evaluated rhs expression.

```
// a single-owned basic assignment
forall (i = _, j = _ on P(block(i,m),2,*))
  ... = ...

// a regular two-owned basic assignment
forall (i = _, j = _ on P(block(i,m),2,*), C.P(block_cyclic(i,m,p)))
  ... = ...

// an semi-regular two-owned basic assignment (scatter)
forall (i = _, j = _ on P(block(ind(i),m),2,*), C.P(block_cyclic(i,m,p)))
  ... = ...
```

Example 11: Examples of basic assignments.

A dimension of an owner is called regular if the expression used as the first argument in the distribution function is linear in one of the loop variables of the basic assignment, otherwise it is called irregular. An owner is called irregular if one of its dimensions is irregular otherwise it is regular.

For example the receiving owner of the second assignment in Example 11, i.e. \(P(block(i,m),2,*))\), is regular in the loop variable \(i\). The receiving owner of the third assignment, i.e. \(P(block(ind(i),m),2,*))\), is irregular in its first dimension because \(ind(i)\) is not a linear function in the loop variable \(i\).

The following basic assignments can be evaluated efficiently:

- **Single-owned assignment**: An assignment is called a single-owned assignment if the lhs expression, the rhs expression, and the mask expression form a single-owned scope. In this case, the processors specified by the owner can enumerate his/their own local iteration space. The more dimensions of the owner are regular, the more efficient the compiler can
enumerate the iteration space. Single-owned assignments can be executed without communication.

- **Two-owned assignment**: An assignment is called a two-owned assignment if it consists of two single-owned scopes, one for the sending part and one for the receiving part. Two-owned assignments will result in communication between the sending-owner and the receiving-owner.

There are three subclasses of two-owned assignments.

- A **regular** two-owned assignment has two regular owners.
- A **gather assignment** has an irregular sending owner and a regular receiving owner.
- A **scatter assignment** has a regular sending-owner and an irregular receiving-owner.

The gather and scatter assignments are called **semi-regular assignments**. For all three basic communication assignments (regular, gather, and scatter) an efficient enumeration scheme can be generated for the send and the receive set of iterations for a given processor pair. In case both the lhs and rhs expressions have different irregular owners, the assignment is split into two semi-regular assignments by introducing a properly aligned regular temporary array.

The exact translation of a basic assignment into a sequential SPMD program is the task of the SPMD engine. Details about the SPMD engine are given in Section 4.4.4.

### 4.3.3 Mapping application and mapping extraction

Mappings and owners are closely related. The compiler can derive an owner from a mapping when a mapped array is referenced in an expression, as shown in Example 12. The formal arguments of the mapping of A, i.e. \((i,j)\) are substituted with the actual arguments used in the subscript expression of \(A\), i.e. \((4,6)\). The resulting owner after mapping application is \(P(block(4,m),*)\). The derivation of owners from mappings is called **mapping application**.

```
A: mapped (i,j) on P(block(i,m),*)  // From Mapping to Owner
    A(4,6)

→

A: mapped (i,j) on P(block(i,m),*)

  on P(block(4,m),*) A(4,6)  // derived from the mapping of A
```

Example 12: Owner application: from mappings to owners.

The inverse operation is also possible and is called **mapping extraction**. The compiler can extract a mapping from an owner as shown in Example 13. Because a mapping has a stricter format
than an owner does, a direct mapping extraction is not always possible. For mapping extraction, we need to distinguish two types of owners: mappable owners and non-mappable owners.

```
forall (i = _, j = _)  // From Owner to Mapping
    _ on P(block(3*i,m),*,cyclic(i*j,p),k) { _ }  
→

A_{re} \quad mapped (i,j) on P(block(3*i),*,*,k)  // derived from a generalization  
    // of the owner

forall (i = _, j = _)
    A_{re}(i,j) = on P(block(3*i,m),*,cyclic(i*j,p),k) { _ }
```

Example 13: Mapping extraction: from owners to mappings.

An owner is mappable in \((i_1, ..., i_n)\), if there exists a mapping specification of the format as given in Figure 6 with \((i_1, ..., i_n)\) as formal parameters, i.e. when a mapping specification can be extracted directly from an owner.

A non-mappable owner can be generalized into a mappable owner by replacing the non-linear alignment functions with * distribution functions (see the `forall` in Example 13). Of the remaining linear alignment functions duplicates using the same loop variable must be replaced with * distribution functions.

Mapping extraction will become important when the compiler needs to introduce mapped temporary arrays to hold values from mapped expressions.

### 4.4 The Compilation System (CoSy)

A CoSy compiler is built from a collection of engines, each performing a specific task on the intermediate representation (IR) of the program, which is stored in a common data pool. A typical compiler contains a front-end engine to translate source files to the IR, various analysis and optimizer engines operating on this IR and a code generator translating the IR to assembly code.

The basic building blocks of a compiler are the engines. Each engine implements a transformation algorithm (e.g. common sub-expression elimination), independent of the scheduling of other engines in the compilation system. The engines can therefore be shared, in source or binary form, between several compilers in the same framework, and between several frameworks, at several sites as well. To support this modularity, the framework provides separate name-spaces for the engines. An engine defines its private domains and the view it has on the IR. Parts of the engines can be generated by engine generators, like BEG (back-end generator), the code generator-generator system.
The CoSy compiler system as compiler generation system offers the functionality to define a private **Medium-level Intermediate Representation** (MIR). However, when it comes to writing a compiler consisting of several engines, standardization on a MIR like CCMIR, a Common CoSy MIR, is required, exactly because it defines their common knowledge of the intermediate representation. The source languages supported by the CCMIR are ISO C (optionally with DSP extensions), K&R C, Fortran 77, Fortran 90, Fortran 95, HPF, Pascal, and Modula-2. All CoSy front-ends produce code in CCMIR format.

The CCMIR exists of several domains specifying Types, Objects, Expressions, Statements, Basic Blocks, and Units. The collection is structured as a multi-graph with those basic entities as nodes. Each node contains **attributes** that further detail its meaning. All contextual references in the source program have been resolved by the front-ends and are represented in the graphs as **references** to other nodes.

The root of the internal representation IR is the unit (**mirUnit**). It forms a separate compilation unit (e.g. source files, modules). Each unit contains a list of all types (**mirTYPE**), global variables (**mirGLOBAL**), and procedures (**mirProcGlobal**) specified inside the unit. Each procedure has a body (**mirBody**), which is either empty (**mirNoBody**) or as specified in the unit. A body consists of a list of local variables and a graph of basic blocks (**mirBasicBlock**). A basic block consists of a list of statements (**mirSTMT**), and statements contain expressions (**mirEXPR**). Each variable and expression is typed by referencing to a **mirTYPE** from the unit's type list.

In the CCMIR model, a user program manipulates objects (variables), by executing statements (like IF ... THEN and GOTO) that contain expressions. The result of an expression is a value of some type (e.g. signed integer) and can be stored in an object of that type.

Our CoSy HPF compilation system contains several engines (see Figure 7).
One of these engines is the HPF front-end, which parses the HPF source code and fills the IR. Other engines are classical optimizers such as invariant code motion (ICM) and common sub-expression elimination (CSE), and vectorizers (Vector). These engines are called classical engines because they do not need to have a notion of processor locations (mapping and owner attributes). Engines that do have this notion are called owner-aware engines. The owner-aware engines are:

- The Align-engine, which attributes non-attributed arrays with a mapping and non-attributed operators, statements, basic blocks, and procedure bodies with owners.
- The Decompose-engine, which decomposes multi-owned assignments into basic assignments by introducing, mapped temporary variables.
- The SPMD-engine, which lowers a basic assignment (see Section 4.6) into Single Program Multiple Data code by transforming it into a sequence of packing, sending, receiving, and assigning code fragments.

The following design criteria have been used for the owner-aware engines:

- By disabling the owner-aware engines, the compiler must act as a Fortran 95 compiler.
- All engines act on a medium level IR to avoid language specific dependencies, and to allow other front-ends with a similar HPF data-parallel model to be added to the framework.
• All transformation engines act on a scope as small as possible and can be called in any order to allow advanced optimization schemes.

• The IR may be extended with data-parallel attributes, but existing classical engines do not need to be aware of these attributes.

• The number of IR-extensions needs to be as small as possible.

The owner-aware engines are normally used in the order as given above. They can however be interspersed with classical engines. The next sections will discuss the owner aware engines in more detail.

4.4.1 The HPF Front-end

The HPF front-end is essentially a sequential F95 front-end which stores all HPF specific information as annotations into the IR. The HPF run-time information is mainly stored into three run-time descriptors, a **processor descriptor**, a **mapping descriptor**, and an **array descriptor**.

HPF data mapping information becomes a normalized mapping attribute in the IR typing system. For each distincte mapped array type an **array descriptor** is constructed and initialized with the array type information, like the local lower and upper bounds. Each array descriptor contains a pointer to its associated mapping descriptor.

From the set of mapped arrays, a (normally smaller) set of mappings can be derived. For each mapping, a **mapping descriptor** is constructed, and initialized with the alignment and distribution information. Like the alignment stride and offset, the distribution type (**const**, *, **block**, **cyclic**, **block_cyclic**), and the blocksize \( m_0 \) and cycle length \( m_l \), if needed. Each mapping descriptor contains a pointer to its associated processor descriptor.

For each HPF PROCESSORS declaration, a **processor descriptor** is constructed and initialized with the processor shape. The one-dimensional **processor index vector** \( P.me \), which corresponds to the processor id \( pid \), is determined and stored into the processor descriptor.

A special processor array \( P_{world} \) is globally defined. It contains all SPMD processes and has a shape as given at startup-time. All other processor arrays can be derived from \( P_{world} \).

Multi-dimensional array subscripts are generated as a list of one-dimensional CCMIR subscripts. Scalar assignments and volume assignments, like Array assignments, WHERE, FORALL statements and FORALL constructs, are stored as a normalized volume assignment (See Section 3.4.1). All intrinsic subroutines, array reduction operations and inquiry functions are inlined.

When the sample HPF input of Example 14 is fed to the HPF Front-end, it will generate an internal representation (shown in a pseudo C language), as given in Example 15.
Example 14: Sample HPF input

typedef int T0[N_tmp]
   mapped (i) on map_D->proc[block_cyclic(N_tmp*i+3),
                        Ddum->map.dim[0].m,
                        Ddum->map.dim[0].ml],*];

typedef T0   T1[M];
typedef TSc  int mapped () on _;
typedef int Tdum0[Ddum->dim[0].ext] mapped (i) on _;
typedef Tdum1[Ddum->dim[1].ext] mapped (i) on _;

void SUB(Tdum1 *Dum, Thpf *Ddum, int N, int M)(
   int   N_tmp;
   T0    *A;
   T0    *D;
   Tmap  *map_D;
   Thpf  *Da,*Dd;
   TSc   *Sc;

   N_tmp = N;
   map_D = perts_map(Pworld,1,N_tmp,3,cell(M/Pworld->dim[0].ext),block,1);
   Dd = perts_Dhpf(map_D,2,1,N_tmp,1,M);
   Da = perts_Dhpf(map_D,2,1,N_tmp,1,F);

   D = (TD2*)perts_malloc(Dd.size);
   A = (TA2*)perts_malloc(Da.size);

   N = 10;
   - A[i][j] -
   perts_free(A,D);
}

Example 15: The corresponding IR of the HPF sample of Example 14.

In the generated code, the subroutine SUB gets an extra parameter. The extra parameter Ddum is a run-time descriptor that travels along with the parameter Dum. The subroutine that is calling
The compilation system wants to compare mappings symbolically, all expressions used in the IR typing system (the typedef's) must be made invariant\textsuperscript{25}. This can be done by introducing a temporary variable $N\_tmp$ to hold the value of $N$ at the entrance of the subroutine. Because CCMIR models multi-dimensional arrays as arrays of arrays all mappings are specified per dimension.

The type of the second dimension of variable $A$, named $T1$, has no owner attribute because it is a collapsed dimension. The first dimension of $A$, named $T0$, however is mapped onto a processor dimension.

The run-time support function, $pepts\_map$ is used to initialize a mapping descriptor. The run-time support routine, $pepts\_Dhpf$ initializes a run-time array descriptor. The run-time support routines, such as $pepts\_map$, $pepts\_Dhpf$ on which the HPF front-end relies are described in Section 4.4.6.

Once the run-time descriptors are initialized, the local variables $A$ and $D$ can be allocated with the run-time support routine $pepts\_malloc$. When the subroutine exits, the local variables $A$ and $D$ are freed again.

### 4.4.2 The Align engine

Once the HPF front-end has parsed the HPF source, a complete CCMIR representation is built and the IR typing system is attributed with mappings and statements are attributed with owners, as specified by the programmer. However, all local variables\textsuperscript{26} without a mapping specification and compiler introduced temporary variables\textsuperscript{27}, still have an unknown mapping. The statements and statement blocks not attributed with the ON directive and operators inside expressions still have unknown owners.

It is the task of the align-engine to find a proper mappings and owners for these IR-nodes, such that data locality is most optimal and work distribution is aligned properly with the data processed. The analysis needed to find proper work and data distributions for the non-specified parts of the program is further explained in Section 4.5.

\textsuperscript{25} Invariant expressions will always evaluate to the same value as long as the expression is used.

\textsuperscript{26} We will only consider local variables as targets for which the align-engine has to select an owner, because the complete usage of the variable can be analyzed. Furthermore, we do not want to create side effects by modifying types of variables defined elsewhere.

\textsuperscript{27} Because the CoSy HPF compiler also consists of engines not aware of owners, it might introduce a temporary variable (with an unknown mapping) because of some sequential optimization (i.e. common sub-expression elimination or vectorization).
4.4.3 The Decompose engine

The decompose-engine is started after the align-engine has found proper owners for each operator in an expression tree. At places where the owner of the operator is not a subset of the owners of its children, a so-called owner conflict occurs.

It is the task of the decompose-engine to transform multi-owned assignments into a sequence of single and two-owned assignments. For each owner conflict in the multi-owned expression tree, a two-owned assignment is introduced by the decompose-engine.

The child expression is replaced with a temporary variable, which has a mapping derived from the owner of the parent (see Section 4.3.3). The temporary variable is initialized with the child expression just before the assignment (as shown in Example 16).

---

A: mapped (i) on P(cyclic(i))
B: mapped (i) on P(block(i))
C: mapped (i) on P(block_cyclic(i))

// a multi-owned assignment
forall (i = ... on P(block(i)), P(cyclic(i)), P(block_cyclic(i)))
  A(i) = A(i) + on P(block(i))[B(i) * on P(block_cyclic(i))\{C(i)\}]  
  \rightarrow 
A: mapped (i) on P(cyclic(i))
B,T: mapped (i) on P(block(i))
C: mapped (i) on P(block_cyclic(i))

forall (i = ... on P(block(i)), P(block_cyclic(i)))  // a two-owned assignment
  T(i) = on P(block_cyclic(i))\{C(i)\}
forall (i = ... on P(cyclic(i)), P(block(i)))  // a two-owned assignment
  A(i) = A(i) + on P(block(i))[B(i) * T(i)]

---

Example 16: Decomposition of a multi-owned assignment.

Some two-owned assignments do get decomposed, if the communicated volume can be reduced as shown in Example 17.
Example 17: Reduced communication by proper decomposition.

The shape of the newly introduced temporary variable is derived from the iteration space of the assignment, and the loop variables actually used in the child expression. Example 17 demonstrates that the iteration space of the newly introduced assignment can be smaller than the original one. The sub-expression $B(i)$ does not depend on the iteration variable $j$. The associated volume of sub-expression $B(i)$, which is $I_{ext} \times J_{col}$ of the complete assignment. In this case, the decompose-engine can derive a one-dimensional iteration space from the original two-dimensional iteration space of the assignment, by using a one-dimensional temporary variable.

Example 18: Generalizing an owner.
Even in case of irregular owners the decompose-engine can still find an appropriate mapping for the newly introduced variable $T(i_{ext})$ as shown in Example 18. The align-engine has selected a solution, which puts the owner conflict between the * operator and the $B(i)$ sub-expression in the top assignment, as can be seen by the placement of the on attribute. The communicated volume crossing the owner conflict is now just one-dimensional. If the align engine had placed the owner conflict between the + operator and the $j * B(i)$ expression, to communicated volume would be two-dimensional which is in this case always larger.

The owner of the * operator, i.e. $P(block(i*i))$, is not a mappable owner and therefore cannot be used directly to construct a mapping for the temporary variable $T$. However, the process of owner generalization (see Section 4.3.3) can be used to derive a mappable owner $P(*)$, which can be used as a mapping for $T$.

4.4.4 The SPMD engine

The SPMD engine needs to remove all owner-attributes by lowering them into a sequential program following the SPMD programming model. There are three cases the SPMD engine needs to handle; owner attributed IR nodes with a scope, accessed mapped variables, and basic assignments (single-owned or two-owned assignments).

The first two cases will be discussed in the next two sections. The lowering of basic assignment is the most crucial part of the compilation system and can be very complex. Therefore, it will be discussed separately in Section 4.6.

Owner attributed IR nodes with a scope.

Owner attributed IR nodes with a scope (statements, basic blocks and procedure bodies) can simply be transformed into SPMD code by surrounding the IR node with an if statement with a special mask (see Example 19).

```plaintext
on P1(block(i)), P2(cyclic(j)){
    ...
}

⇒

if (P1(∅) == P1(block(i))) and P2(∅) == P2(cyclic(j)){
    ...
}
```

Example 19: Lowering of an owner attributed scope.

---

26 After the align engine all remaining owner annotations in expressions coincide with owner conflicts.
The mask checks if the owner evaluates to the processor index vector $P(\theta)$ that is stored inside the processor descriptor. Each processor will have its own value of $P(\theta)$. The resulting processor set that will enter the scope is called the active processor set for that scope.

### Accessing mapped variables

For all array subscript operators (*mirSubscript*) with an owner-attributed array-type (*mirArray*), the SPMD engine needs to introduce proper global-to-local functions. A **global-to-local function** translates a global index (pre-SPMD) into a local index (post-SPMD) that can be used to reference local memory. A special run-time support function $g2l$ (see Section 4.4.6) is provided to simplify the generation of SPMD code (as shown in Example 20).

```plaintext
int A[l:N][l:M] mapped (i,j) on P[block(i),*];

- A[i][j] →

- A[g2l(_,i)][j] →
```

Example 20: Usage of the global-to-local support function.

### 4.4.5 Parallel IR Extensions

So far, we only have mentioned how to deal with mappings and owners, without telling the internal representation of these attributes. All extensions to the Internal Representation (IR) needed for our data- and task-parallel model are prefixed with *pir* (Parallel IR). As mappings and owners are tightly related to each other, both are expressed as *pirLocations* (see Figure 8). An owner is a simple location expression, while a mapping is a parameterized location expression.

A mapping attribute is a *mirType* attribute, labeled ‘Mapping’. Because an owner attribute can attribute an expression, statement, or basic block, each of these domains is extended with an owner. The owner attribute is labeled ‘On’.

A *pirOwner* simply consists of a reference to a processor object and an index (a list of *mirEXPR* expressions). The *mirEXPR* domain is extended with a *pirAllExpr* operator, for the specification of replications in the owner index.
The expressions used in the pirLocation must be invariant expressions (i.e. the complete expression must always evaluate to the same value independent where it is evaluated inside the control flow). A pirAllExpr is an invariant expression.

At the highest IR level (the mirUnit) all pirLocation’s are registered in a linear list for easy manipulation in owner aware engines. All other pirLocation’s are references in this location list. pirMappings are shared between mapping attributes and pirOwners are shared between owner attributes whenever possible.

Engines not aware of the PIR extensions will create by default a nil pointer, which is denoted by the initop(0) property for each field.

### 4.4.6 Parallel Engines Run Time System (PERTS)

The Parallel Engines Run-time System (PERTS) provides the necessary additional types, global data, and functions to facilitate the task- and data-parallel translation process of the compiler. The support for the HPF PROCESSOR directive is straightforward (see Figure 9).

---

29 Expressions can be made invariant by storing the value of the expression in write-once variables, initialized at the point of declaration.
typedef struct {
    int extent, me, stride;
} Tproc_dim;

typedef struct {
    int rank;
    Tproc_dim *dim;
    Bool i_am_member;
    Bool i_am_master;
    Comm replicates_comm;
    Comm owner_comm;
} Tproc;

extern Tpid pid
extern Tproc *PWorld;

Tproc *perts_proc(Tproc *Parent, int rank, ...); // (lwb, upb, str)

Figure 9: PERTS processor declaration support

The function perts_proc creates, initializes, and returns a processor run-time descriptor. It needs an existing processor run-time descriptor Parent, the rank of the parent processor array rank, and a list of (lwb, upb, str) 3-tuples describing the section of the parent of which it consists. Special (lwb, upb, str) 3-tuples are used. A (c, c, I) 3-tuple is used to specify a single point and a (-I, -I, -I) 3-tuple to specify a replicated dimension.

The structure Tproc holds the run-time information. It contains the rank of the processor array in rank, and information about the dimensions in dim. All dimensions of a processor run-time descriptor are zero based. For each dimension, extent is the extent of the dimension, and stride is the stride the pid would make when this processor dimension is incremented by one.

For those SPMD processes with a pid that does correspond to a processor index, the field me is calculated such that the inner product of me and stride over all dimensions equals to the processor pid. In that case, the flag i_am_member is set to true. For those SPMD processes with a pid that does not correspond to an index of the processor array will have a me coordinate equal to -I in each dimension and the flag i_am_member is set to false.

The two comm-typed variables replicates_comm and owner_comm are two communicators. A communicator is a set of processors pid's amongst which collective communication can take place. They closely resemble MPI [51] communicators.

The communicator replicates_comm contains all SPMD processors, which have the same processor index. One of the SPMD processors from the replicates_comm communicator will have its i_am_master flag set. That processor is called the master processor, or simply the master, associated with the processor index, and will be responsible for sending information, when the master-sends-rule is used (see Section 4.6.1).
The communicator *owner_comm* contains all SPMD processors that can possibly be specified with an processor index (i.e. the \(P^{(\ldots)}\) processor set), and have their *i_am_master* flag set.

For the predefined processor array \(P_{\text{world}}\), its *replicated_comm* communicator contains a single member (the processors own *pid*), its *owners_comm* communicator contains all SPMD processes as specified at startup-time. The *i_am_master* and *i_am_member* flags are set.

The PERTS support for the mapping information of HPF mapped objects is given in Figure 10.

```c
typedef enum {Block, Cyclic, Block_cyclic, Replicated, Constant} Tdist;

typedef struct {
    int   a,b,m0,m1,g_dim;
    Tdist dtype;
} Tmap_dim;

typedef struct {
    int    rank;
    Tmap_dim *dim;
    Tproc  *proc;
} Tmap;

Tmap *perts_map(Tproc *proc, int rank, ...); // (a,b,m,dtype,g_dim)...
```

*Figure 10: PERTS mapping declaration support*

The function *perts_map* creates, initializes, and returns a mapping descriptor. It takes as arguments the processor run-time descriptor *proc* onto which is mapped, a *rank* specifying the number of 5-tuples \((a, b, m_0, m_1, dtype, g\_dim)\), denoting the alignment stride, the alignment offset, the block size, the cycle length, the distribution type, and the corresponding global dimension of the mapped array respectively.

The specified rank must be equal to the processor rank of *proc*. Some special coding is used to describe a constant and replicated distribution. A \((0, c, 0, Constant, 0)\) 5-tuple describes a constant distribution. A \((0, 0, 0, Replicated, 0)\) 5-tuple describes a replicated distribution.

The PERTS mapped variable support is almost the same as ordinary Fortran 95 run-time array descriptor or dope vector support, except that an additional pointer is added, which points to the mapping run-time descriptor (see Figure 11).
typedef struct {
    int lwb,ext,str;
  } Thpfar_dim;

typedef struct {
    void    *addr;
    int     rank;
    Thpfar_dim *dim;
    Tmap    *map;
  } Thpf;

Thpfar *pers_Dhpf(Tmap *map, int rank, ...); // (lwb,upb)

Figure 11: PERTS mapped variables support

The global-to-local run-time support function g2l that the SPMD-engine inserts in subscripts of mapped dimensions of an array and the loop-bounds reduction functions, lus1, lus21, and lus22 are given in Figure 12.

int g2l(Tmap_dim map[], int g);
int g2p(Tmap_dim map[], int g);

int lus1 (int cnt, int a, int b, Tmap_dim *md, int *i_l, int *i_s);
int lus21 (int cnt, int a, int b, Tmap_dim *md, es_t *es,
          int *i0_l, int *i0_s, int *u_l, int *u_s);
int lus22(es_t *es, int i0, int u, int *i_l, int *i_s);

Figure 12: PERTS support functions

The exact use of the loop-bounds reduction functions is given in Section 5.2.1.

4.5 Alignment analysis

The alignment-engine is responsible for the proper placement of work such that the accessed data is locally available as much as possible. The analysis that is needed to fulfill the above requirement is described below.
Assume an expression node as given in Figure 13. The plus operator has two children $A$ and $B$ each child with their own owner $Oc_1^{30}$ and $Oc_2$. The plus operator itself is used twice in different expressions$^{31}$, each parent having its owner $Op_1$ and $Op_2$. Whatever the align-engine selects as an owner for the plus operator it will always conflict with at least one of its children or its parent's owner. Once an owner is selected for the plus operator owner conflicts will occur.

When evaluating the expression according to the owner attributes, only at owner conflict-positions data will flow from one owner (processor location) to the other. This involves communication which is very expensive, on both distributed-memory machines and shared memory machines (e.g. the data in cache can not be reused), compared to data flowing locally on the same processor.

Therefore, for a given scope, like an expression, statement, or basic block, the best solution is the one, which minimizes the total number of owner conflicts inside a scope and the communication volume involved.

The algorithm that the align-engine uses consists of two phases. In the first phase (the analysis phase), **fixed**- and **variable-owned regions** (sub-graphs) are identified. For each variable owner a

---

$^{30}$ We will not describe the owners in detail, only a name is given. This will allow us to compare owners for equality based on their names only.

$^{31}$ Common sub-expressions can have multiple uses and therefore multiple parents.
selection list is constructed. In the second phase (the selection phase), a fixed owner is selected from the selection list for each variable-owned region.

4.5.1 The alignment analysis phase

In the alignment analysis phase, the fixed and variable owned regions are identified. Each operator must have an owner attribute. These owner attributes can be either a fixed attribute or a variable attribute. A single owned region can have a known fixed owner (denoted with a capital O), or it can have a variable owner (denoted with a capital V).

Fixed owned regions occur at the leave nodes of an expression (e.g. numerical constants). Variable owned regions might occur at intermediate expression nodes. For variable owners an owner selection list is constructed consisting of a list of owners from the bounding regions.

In a depth-first walk over the expression trees, the align-engine tries to find the minimal list of owners involved for the evaluation of the operator. This process is called parent owner derivation. Parent owner derivation uses three kinds of owner derivations: owner reduction, owner generalization and owner privatization.

Depending on the operator one or more of these owner derivations are needed. For instance, a plus operator (+) only needs owner reduction to determine the owner of the plus operator from the owners of its children. For a subscript operator ([ ]) all three owner derivations are needed.

```
Input: child owner list co_l
Output: reduced owner list ro_l

while size(co_l) > 1 do
  o = head(co_l)
  found_overlap = false
  for each element co of co_l do
    if o \cap co \neq \emptyset then
      found_overlap = true
      replace o \cap co with o in co_l
    fi
  od
  if !found_overlap then
    add o to ro_l
  fi
od
add co_l to ro_l
```

Figure 14: Owner reduction algorithm.

The algorithm used for owner reduction is given in Figure 14. The algorithm tries to find the minimal number of non-overlapping owners that are involved in processing the data located at the children of the operator. As input the child owner list co_l is used. The head of the list o is
checked for an overlap with the rest of the list \( co_1 \). If an overlap can be found \( o \cap co_1 = \emptyset \) the element from the child owner list is replaced with the overlap itself. If no overlap can be found the element \( o \) is added to the reduced owner list \( ro_1 \).

For example, a plus operator having a child owner list \( co_1 = \{ P(block(i), *), P(*, block(j)) \} \) can be evaluated at the reduced owner list \( po_1 = \{ P(block(i), block(j)) \} \), without communication.

![Diagram](image)

Figure 15: Deriving a parent owner by owner reduction.

If the resulting parent owner list contains only one element then the operator has a fixed owner. Otherwise, it is a variable owner where the parent owner list will be used as the selection list in the selection phase.

**Owner generalization** is needed in the derivation of the owner of index expressions in a subscript operator and enlarges the set of processors involved. If a mapped expression is used as an access function of the subscript operator \([\ ],\) like \( A[ind[i]] \) where both \( A \) and \( ind \) are mapped, all processors which own \( A[\star] \) must know the value of \( ind[i] \) just to know if they are the owner of \( A[ind[i]] \). The value of the mapped expression \( ind[i] \) must be broadcast to the owners of \( A[\star] \). Therefore, the owner of the access function is forced to be a **generalization** of the owner of the subscript operator, simply by replacing the access function with a star access function. General-
ized owners have the same processor name and index as the owner from which they are generalized, except that they are superscripted with a star in the dimension in which they are generalized.

**Owner privatization** is needed in the derivation of the owner of the base expression of a subscript operator. It specifies a location where the value of the expression is private to this location; i.e. the expression may evaluate to a different value for each location. An owner \( P(I) \) can be privatized on \( I = (i_1, \ldots, i_n) \) by replacing all owner dimensions containing a variable from \( I \) with the corresponding processor index in that dimension. The corresponding processor index is denoted with the @ operator. For example \( P(i) = P(block(i), cyclic(j)) \) is privatized into \( P(@) = P(@, cyclic(j)) \) which is syntactic sugar for \( P(@) = P(P.me(0), cyclic(j)) \), where \( P.me \) is the processor index vector.

![Diagram](image)

Figure 16: Deriving owners by owner generalization and owner privatization.
In Figure 16, an example is given of owner generalization and owner privatization. The owner of the subscript operator is found by mapping application of the mapping found in the mapping attribute of its type. The owner of the base expression is found by applying owner privatization of the parent owner. The base expression $A$ is already available at each processor $P(@)$ therefore no owner conflict occurs along this edge. The index expression $ind[i]$ must be available at the generalization of the parent owner, i.e. $P(*)$. The index expression itself however is only available at $P(block(i))$ (also found by a mapping application). Because $P(*)$ is not a subset of $P(block(i))$ a owner conflict occurs along the index edge.

Applying these owner derivations for each operator, a complete attributed expression tree results with either fixed or variable owned regions.

\[
A: \textbf{mapped} (i) \text { on } P(block(i))
\]
\[
B: \textbf{mapped} (i) \text { on } P(cyclic(i))
\]

\[
V_2 = \{O_1,V_1\}
\]

\[
V_1 = \{O_1, O_2, V_2\}
\]

\[
O_1 = P(block(i))
\]

\[
O_2 = P(cyclic(i))
\]

\[
A \[ i \]
\]

\[
B \[ i \]
\]

Figure 17: Constant and variable owners of $A(i) = A(i) + B(i)$.
For example the scalar assignment $A(i) = A(i) + B(i)$ can be split into constant- and variable-owned regions as given in Figure 17. The leave nodes $A$ and $B$ (the base addresses of the arrays) are privatized expressions and $i$ is **totally replicated**. The compiler will detect that both sub-expressions $A(i)$ have the same owner $O_i = P(block(i))$ in a similar way as common sub-expressions are detected (see [1] for common sub-expression detection).

Variable owners have an option list consisting of a list of child owners followed by a list of parent owners, which are bounding the operator.

### 4.5.2 The alignment selection phase.

Once all the possible options for each variable owner are identified, the optimal **owner solution** must be found which minimizes a chosen assignment cost function. A cost function could be for example, the number of edges with owner conflicts weighted with the data volume flowing over these edges. Two different search algorithms will be discussed: a **full search algorithm**, which can find the optimal solution, and a **two-sweep algorithm**, which might result in a sub-optimal solution, with respect to the assignment cost.

A **full search algorithm** can be used to enumerate all possible owner solutions, measure the total cost and select the solution with the minimal cost. Unfortunately, the number of enumerations can easily explode. The number of enumerations is roughly of order $O(c^n)$ where $c$ is the average number of possible options per variable owner and $n$ the number of variable owners. It is, however, quite easy to calculate the exact number of enumeration’s that is needed. If not too large, the full enumeration algorithm can be used.

A **two-sweep algorithm** can be used when a full search is not applicable or not needed. An order $O(n^2)$ algorithm first sweeps the variable owner list bottom-up to attribute each node with a restricted list of candidates and then makes a top-down sweep to cut tie breaks. This algorithm finds the optimal solution as long as no variable owner is used more than once in another option-lists. The graph will then reduce into a tree where a two-sweep algorithm is sufficient.

Turning back to Figure 17, the optimal solution would be $V_I = V_2 = O_I$ resulting in only one owner conflict between the $+$ operator and its right child $B(i)$ (see Example 18).
A: mapped (i) on P(block(i))
B: mapped (i) on P(cyclic(i))

Figure 18: A(i) = A(i) + B(i) after owner selection

4.6 Lowering Basic Assignments

4.6.1 Assignments and communication

Even a simple scalar assignment like $x = y$ can result in very different communication patterns between the processor(s) owning the scalar $y$ and the processor(s) owning the scalar $x$ as is depicted in Figure 19.

32 Although the object is a scalar, the HPF mappings allow, through replication, a set of processor to be the owner of the same scalar. Each instance of this scalar must store the same value when referenced.
The processors owning both the right-hand-side (rhs) expression and the left-hand-side (lhs) expression can execute the assignment locally by reading from local memory and storing in local memory. This is called a local-to-local copy. Let $P_{recv}$ be the set of processors owning the lhs. Similarly, let $P_{send}$ be the set of processors owning the rhs. To get the rhs information onto $P_{recv}$ one (or several) members from $P_{send}$ must send it to each member of $P_{recv}$.

If only one processor sends the information, it is called the master sends rule. The processor responsible for sending the information is called the sending master. The processors owning the rhs expression are called the owner\(^{33}\) of the rhs expression. If only one processor receives the information and then multicasts the information to the processor set that is the owner of the lhs, it is called the master receives rule.

The processor responsible for receiving the information is called the receiving master. The receiving master is responsible for the replication (multicast) of the receive buffer amongst the

\(^{33}\) Although the owner may be a set of processors, a singular noun is used.
owners of the lhs. Depending on the network topology, different kinds (master or owner) of sending and receiving rules may be used.

When using mapped arrays in a volume assignment, the same problem as for a scalar assignment shows up. In fact, it is even more complicated because the owners of the lhs, rhs, and mask expressions must be considered for the complete iteration space.

### 4.6.2 More about locations

Before we can continue explaining the lowering of basic assignments, we need to introduce some notation conventions.

A processor set specification can contain two kinds of replication: wildcard replications and element replications. **Wildcard replications** are specified as wildcards in a processor array and are known at compile-time. **Element replications** are the replications of each element of the processor array and are known at run-time.

For instance, let processor array $P$ be a regular section of $P_{\text{world}}$ (e.g. $P(p) \rightarrow P_{\text{world}}(2, *, p_{1}, p_{2})$), where $p = (p_{1}, p_{2})$. The $*$ acts as a wildcard to select a range of $P_{\text{world}}$. In this case $P(0,0)$ still denotes a set of known processors, because we know how it is derived from $P_{\text{world}}$. In some cases, where $P$ is inherited, it is not known to the compiler whether $P(0,0)$ refers to a single processor or a processor set.

A master of a processor set $P$, $\text{Master}(P)$, is a single member of the processor set. When the master operator contains a wildcard replication, the compiler is free to replace it with a proper value. It thereby resolves the compiler known part of the replication. The remaining element replication is resolved at runtime by using the $\text{i\_am\_master}$ flag in the run-time descriptor (see Section 4.4.6).

A special notation $@$ can be used in a dimension of a processor array. When used in the $i$-th dimension of a processor array $P$, it evaluates to the $i$-th element from the processor index vector, i.e. $P_{\text{me}}[i]$.

For example, if an SPMD program is started within a group of $np = 6$ processors and labeled with processor id $\text{pid} = 5$ (see Section 2.4), and $P_{\text{world}}$ is specified as $2 \times 3$, then its processor index vector is $P_{\text{world}}[\text{me}] = (1, 2)$. $P_{\text{world}}(i, @)$ would evaluate to $P_{\text{world}}(i, P_{\text{world}}[\text{me}]) = P_{\text{world}}(i, 2)$.

Scanning a processor set is denoted as ‘do $p \in P$ ... od’ and consists of a do-loopnest scanning all elements $p$ of the processor set $P$. The loop variable $p$ is printed bold to denote a vector. With $P(p) \rightarrow P_{\text{world}}(2, *, p_{1}, p_{2})$, $p = (p_{1}, p_{2})$ would scan $P_{\text{world}}(2, *, *, *)$ with a do-loopnest of depth two.

---

34 Some networks may favor multiple processors sending the same information to corresponding receiving processors.

35 Counting starts from zero.
4.6.3 Lowering Scalar assignments

Given a scalar normalized assignment\(^{36}\) as shown in Example 21, where the complete assignment is owned by processor set \(P_{\text{recv}}\) except for the sub-expression \(expr\), which is owned by the processor set \(P_{\text{send}}\). The compiler will try to find a relation between \(P_{\text{recv}}\) and \(P_{\text{send}}\), in the order given below.

Relation I: \(P_{\text{recv}} \subseteq P_{\text{send}}\)

```plaintext
forall (on P_{\text{recv}}, P_{\text{send}}) {
    A = A + on P_{\text{send}}(expr)
end forall

=>

on P_{\text{recv}} {
    A = A + expr  // assignment
}
```

Example 21: SPMD code for a scalar assignment when \(P_{\text{recv}} \subseteq P_{\text{send}}\).

First the compiler checks whether, as in Example 21, \(P_{\text{recv}} \subseteq P_{\text{send}}\) holds (e.g. \(P_{\text{recv}} = P(\text{block}(i),\) \(\text{cyclic}(j))\) and \(P_{\text{send}} = P(\text{block}(i), *)\)). If so, then there is no communication needed between the two processor-sets.

All processors from \(P_{\text{recv}}\) can evaluate \(expr\) locally and assign its value to the lhs expression. Lowering a single-owned scalar assignment can be treated as lowering a two owned scalar assignment where \(P_{\text{recv}} = P_{\text{send}}\).

\(^{36}\) Scalar assignments are normalized into foralls with an empty iteration space (see Section 4.2.2).
Relation II: $P_{recv} \cap P_{send} \neq \emptyset$

```
forall ( on P_{recv}, P_{send} )
  A = A + on P_{send}(expr)
end forall

→

on Master(P_{recv} \cap P_{send}){
  buf = expr
}

on P_{recv} {
  broadcast buf from Master(P_{recv} \cap P_{send}) to P_{recv}
  A = A + buf
}
```

Example 22: SPMD code for a scalar assignment when $P_{recv} \cap P_{send} \neq \emptyset$

If relation I does not hold the compiler checks whether, as in Example 22, $P_{recv} \cap P_{send} \neq \emptyset$ holds (e.g. $P_{recv} = P(*,\ cyclic(j))$ and $P_{send} = P(block(i), *)$). If this is the case, then there is a single processor (i.e. Master($P_{recv} \cap P_{send}$) = Master($P(block(i), cyclic(j))$), that is also a member of $P_{recv}$, which can evaluate $expr$. Because it is also a member of $P_{recv}$ it can be the source of a broadcast to the members of $P_{recv}$. This allows, as in MPI\textsuperscript{37} [51], that collective operations (like the multicast) can be called by all members of the destination processor set (i.e. $P_{recv}$).

\textsuperscript{37}The communication primitives are designed that they overlap as much as possible with MPI communication primitives, to make the code more portable.
Relation III: $P_{recv} \cap P_{send} = \emptyset$

```c
forall ( on P_{recv}, P_{send} ) {
    A = A + on P_{send}(expr)
} endforall

=>

on Master(P_{send}){
    buf_{send} = expr              // packing
    send buf_{send} to Master(P_{recv})  // sending
}
on Master(P_{recv}){
    recv buf_{recv} from Master(P_{send})  // receiving
}
on P_{recv} {
    broadcast buf_{recv} from Master(P_{recv}) to P_{recv}  // replicating
    A = A + buf_{recv}  // assignment
}
```

Example 23: SPMD code for a scalar assignment when $P_{recv} \cap P_{send} = \emptyset$

When the compiler can not find the previous two relations between $P_{recv}$ and $P_{send}$, the compiler will fall back to the most general (and most inefficient) method as given in Example 23. The compiler assumes that their is no processor overlap (i.e. $P_{recv} \cap P_{send} = \emptyset$). The master of $expr$, $Master(P_{send})$, will pack and send to the master of the assignment, $Master(P_{recv})$. The master of the assignment will broadcast the receive buffer among the owner of the assignment. The owner of the assignment will do the actual assignment.

In all three cases, the two-owned scalar assignment is lowered into a sequence of phases (labeled with comments in the previous examples) with only single-owned scopes. The only processor set involved in executing the complete sequence is $P_{recv} \cup P_{send}$ (i.e. on $P_{recv}$, $P_{send}$). Subsets of the involved processor set are involved in the execution of parts of the sequence. As the relation between the two owners is increasingly general, more phases appear in the sequence.

4.6.4 Lowering Volume assignments

The lowering of a volume assignment is a bit more complicated than lowering a scalar assignment. Let the iteration volume of a volume assignment be given by ($I = \ldots$), where $I$ is a vector of iteration variables, and where the inner loops bounds may depend on outer loop iteration variables.

The variable $A$ is an array indexed by $I$. The owners $P_{recv}(I)$ and $P_{send}(I)$ now depend on values of $I$ (see Example 24). However, the compiler can still determine which of the three previous mentioned relations between $P_{recv}(I)$ and $P_{send}(I)$ holds, because the symbolic comparison of owners is independent from the iteration volume.
for all (I = ... on P_{recv}(I), P_{send}(I)) 
end forall

→
on P_{recv}(@) { 
do (I | P_{recv}(@) \subseteq P_{recv}(I)) \quad // assignment
  
od

Example 24: SPMD code for a volume assignment when \( P_{recv}(I) \subseteq P_{send}(I) \)

Relation I: \( P_{recv}(I) \subseteq P_{send}(I) \)

First the compiler checks whether, as in Example 24, \( P_{recv}(I) \subseteq P_{send}(I) \) holds (e.g. \( P_{recv}(i, j) = P(block(i), cyclic(j)) \) and \( P_{send}(i, j) = P(block(i), @) \)). If this is the case then no communication is needed between the processors. For each iteration instance \( I \), all processors from \( P_{recv}(I) \) can evaluate \( expr \) locally and assign its value to the lhs expression. The result is a do loop nest with a special mask (a so-called owner mask) in it. The do loop is masked with the privatization of \( P_{recv}(I) \), i.e. \( P_{recv}(@) \) because these are the processors that could possibly be involved in evaluating the assignment. All other processors can skip the packing phase.

For the moment, we will not go into the details of how to actually lower the do loop. The technique is called mask absorption (or loop bound reduction) and will be described later in Section 5.1.2. All we need to know is that the do loop will enumerate all iterations \( I \) for which the receiving processor index \( P_{recv}(@) \) is an element of \( P_{recv}(I) \). In other words it enumerates all iterations which are owned by \( P_{recv}(@) \).

Lowering of a single-owned volume assignment can be treated as lowering a two owned volume assignment where \( P_{recv} = P_{send} \).
Relation II: \( P_{\text{recv}}(I) \cap P_{\text{send}}(I) \neq \emptyset \)

```plaintext
forall (I = ... on P_{\text{recv}}(I), P_{\text{send}}(I))
A[I] = A[I] + on P_{\text{send}}(I)(\text{expr})
end forall

\rightarrow

on Master(P_{\text{recv}}(*) \cap P_{\text{send}}(*))
\begin{align*}
\text{cnt} &= 0 \\
\text{do} \{ I = \ldots \mid P_{\text{recv}}(\emptyset) \in \text{Master}(P_{\text{recv}}(I) \cap P_{\text{send}}(I)) \} & \quad \text{// packing} \\
& \quad \text{buf}[\text{cnt}++] = \text{expr} \\
& \quad \text{od}
\}
on P_{\text{recv}}(*)
\begin{align*}
\text{broadcast} \quad \text{buf}[] & \quad \text{// replication} \\
\text{from} \quad \text{Master}(P_{\text{recv}}(\emptyset) \cap P_{\text{send}}(\emptyset)) \text{ to } P_{\text{recv}}(\emptyset)
\\text{cnt} &= 0 \\
\text{do} \{ I = \ldots \mid P_{\text{recv}}(\emptyset) \in P_{\text{recv}}(I) \} & \quad \text{// assignment} \\
& \quad A[I] = A[I] + \text{buf}[\text{cnt}++] \\
& \quad \text{od}
\}
```

Example 25: SPMD code for a volume assignment when \( P_{\text{recv}}(I) \cap P_{\text{send}}(I) \neq \emptyset \)

If relation I does not hold then the compiler checks whether \( P_{\text{recv}}(I) \cap P_{\text{send}}(I) \neq \emptyset \) holds (e.g. \( P_{\text{recv}}(i, j) = P(*, \text{cyclic}(j)) \) and \( P_{\text{send}}(i, j) = P(\text{block}(i), *) \)). If so, then there is for each iteration a single processor (i.e. \( \text{Master}(P_{\text{recv}}(I) \cap P_{\text{send}}(I)) = \text{Master}(P(\text{block}(i), \text{cyclic}(j))) \)), which can evaluate \( \text{expr} \), and is also a member of \( P_{\text{recv}}(I) \) (the owner of the Lhs expression), as shown in Example 25. Notice the similarity in phases with the scalar assignment.

The packing phase is masked with the privatization of \( \text{Master}(P_{\text{recv}}(I) \cap P_{\text{send}}(I)) \), i.e. \( \text{Master}(P_{\text{recv}}(\emptyset) \cap P_{\text{send}}(\emptyset)) \). These are all processors that could possibly be involved in packing the values of \( \text{expr} \). The do loop of the packing phase will enumerate all iterations \( I \) which are owned by the master of the intersection of the sending and receiving owner, i.e. \( \text{Master}(P_{\text{recv}}(I) \cap P_{\text{send}}(I)) \). The values of \( \text{expr} \) are packed consecutively into the send buffer \( \text{buf} \) for the complete iteration space.

The replication phase will replicate all packed buffers from the location where they are packed, i.e. \( \text{Master}(P_{\text{recv}}(\emptyset) \cap P_{\text{send}}(\emptyset)) \) to the processors that need these buffers, i.e. \( P_{\text{recv}}(\emptyset) \). The assignment phase consists of a do loop filtering out all iterations which are owned by \( P_{\text{recv}}(\emptyset) \). Because only the processors \( P_{\text{recv}}(\emptyset) \) are involved in the replication and assignment phase a single mask can be used around both phases.
Relation III: \( P_{\text{recv}}(l) \cap P_{\text{send}}(l) = \emptyset \)

In case the compiler cannot find the previous two relations between \( P_{\text{recv}}(l) \) and \( P_{\text{send}}(l) \), the compiler will fall back to the most general (and a more inefficient) method as given in Example 26. As can be seen, similar phases occur as in the previous example (packing, sending, receiving, replicating, and assignment).

```plaintext
forall \{ I = \_ on P_{\text{recv}}(I), P_{\text{send}}(I) \}
    A[I] = A[I] + on P_{\text{send}}(I) \{ expr \}
end forall

→

on Master(P_{\text{send}}(*)) \{ // packing
    cnt = 0
    do p ∈ Master(P_{\text{recv}}(*))
        offset_{send}[p] = cnt
        do \{ I = \_ | p ∈ Master(P_{\text{recv}}(I)) \land P_{\text{send}}(\emptyset) ∈ Master(P_{\text{send}}(I)) \}
            buf_{send[count+]} = expr
            size_{send}[p] = cnt - offset_{send}[p]
        od
    od

communicate size information
on Master(P_{\text{send}}(*)) \{ // sending
    do p ∈ Master(P_{\text{recv}}(*))
        if size_{send}[p] != 0
            send buf_{send}[offset_{send}[p]:offset_{send}[p]+size_{send}[p]-1] to p
        fi
    od
	on Master(P_{\text{recv}}(*)) \{ // receiving
    do q ∈ Master(P_{\text{send}}(*))
        if size_{recv}[p] != 0
            recv buf_{recv}[offset_{recv}[q]:offset_{recv}[q]+size_{recv}[q]-1] from q
        fi
    od
	on P_{\text{recv}}(*) \{
    broadcast buf_{recv}[:]
    from Master(P_{\text{recv}}(\emptyset)) to P_{\text{recv}}(\emptyset)
    \}
    cnt = 0 \{ // assignment
    do q ∈ P_{\text{recv}}(*)
        do \{ I \mid P_{\text{recv}}(\emptyset) ∈ P_{\text{recv}}(I) \land q ∈ P_{\text{send}}(I) \}
            A = A + buf_{recv[count+]}[cnt++]
        od
    od

Example 26: SPMD code for a volume assignment when \( P_{\text{recv}}(l) \cap P_{\text{send}}(l) = \emptyset \)
Several new phases (sending sizes, replicating sizes, receiving sizes) are introduced to communicate the sizes of the messages before of the actual messages are communicated (see Example 27).

```
on Master(P_{send}(*)) {
  do p ∈ Master(P_{recv}(*))
    send size_{send}[p] to p
  od
}
on Master(P_{recv}(*)) {
  cnt = 0
  do q ∈ Master(P_{send}(*))
    offset_{recv}[q] = cnt
    recv size_{recv}[q] from q
    cnt += size_{recv}[q]
  od
}
on P_{recv}(*){
  broadcast (offset_{recv}[::], size_{recv}[::])
    from Master(P_{recv}(Θ)) to P_{recv}(Θ)
}
```

Example 27: Communicate the size information

This allows filtering out the communication of empty messages. More importantly, it allows the receiving side to receive the actual messages in an arbitrary order, and still placing the arriving messages in a single receive buffer, sorted by the sending processor index. After the receive-buffer is filled, the actual assignment can take place by reading from the receive-buffer sequentially.

In some cases, it can be profitable to calculate the sizes of the messages that need to be received at the receiving side. This can be done by generating code similar to the assignment phase, but instead of actually unpacking the messages, their sizes are measured instead.

In the packing phase, we could scan for the complete iteration space filtered by an owner mask as shown in the first half of Example 28 (i.e. generating the local iteration set of $P_{send}$). However, the order in which the iterations are filtered out will generate an unordered sequence with respect to the receiving master. Special storage techniques must be used to sort the values of expr per processor, because the number of elements per processor is not known in advance.
\begin{verbatim}
\begin{verbatim}
do \{ I = \_ | P_{send}(\_ \_ ) \in Master(P_{send}(I)) \} \\
\_ \_ \\
\rightarrow \\
do p \in Master(P_{recv}(\_ \_ )) \\
do \{ I = \_ | p \in Master(P_{recv}(I)) \land P_{send}(\_ \_ ) \in Master(P_{send}(I)) \} \\
\_ \_ \\
\_ \_ 
\end{verbatim}
\end{verbatim}
\end{verbatim}

Example 28: Making the destination explicit when packing

If the order of packing within a single processor message is not known at the receiving side, special iteration space information needs to be packed along with the original messages. From this iteration space information, the receiver must be able to find the corresponding point \( I \) in the iteration space.

To avoid all these kinds of problems we will scan the iteration space in a different order as is shown in the second half of Example 28. Although it seems to be much more complex, it avoids the previous mentioned problems. All elements can be packed side by side in one buffer and all elements are sorted by the processor to which they need to be sent. No iteration space information needs to be packed and communicated, the assignment order on processor \( q \) of values received from processor \( p \) is the same as the packing order on processor \( p \) of the values to be send to processor \( q \).

For the moment it is enough to know that when the packing and assignment \texttt{do} loops are specified with an \textbf{identical sequence of masks}, the packing and assignment order is guaranteed to be equal. The exact lowering of the \texttt{do \{ I | \_ \_ \_ \}} loop with multiple owner masks is explained in Section 5.1.2.

\section*{4.7 Related work}

\subsection*{4.7.1 The Owner Concept}

The technique of attributing a program with a \textbf{symbolic owner}, and doing \textbf{symbolic comparison} of owners is new. The fact that the compiler can reason about locality or remoteness of two owners by symbolical comparison, or that it can privatize and generalize owners is therefore hardly documented in literature as a separate issue.

Kamachi et. al. [36] present methods for generating communication on compiling HPF programs for distributed-memory machines. They introduce the concept of an iteration template corresponding to an iteration space. Their HPF compiler performs the loop iteration mapping through the two-level mapping of the iteration template in the same way as the data mapping is performed.
in HPF. Making use of this unified mapping model of the data and the loops, communication for non-local accesses is handled based on data-realignment between the user-declared alignment and the optimal alignment, which ensures that only local accesses occur inside the loop.

This strategy is a limited form of alignment analysis as presented in this thesis. Kamachi et. al. only allow communication based on data realignment. In our terminology, owner conflicts can only occur on edges where the child is a subscript operator. Only a single owner can be selected for the complete assignment. This single owner also has to be a mappable owner. The resulting two-owned assignments all have equally aligned left-hand side expressions, and a simple array subscript as right-hand side expression.

Compared to our method of alignment analysis and decomposition the following conclusions can be drawn. No calculation can take place when packing the rhs expression (i.e. the sender computes rule cannot be used). The same holds for the assignment to the left-hand expression (i.e. the receiver computes rule can not be used). Because the owner of the assignment must be expressed as the application of an iteration template mapping, it has to be a mappable owner. All these restrictions will result in a restricted set of communication patterns that can occur.

When following the method of Kamachi et. al., the best selection that can be found for the owner of the expression is some sort of majority computes rule. The owner of the assignment will be the owner that represents the majority of the owners in the owner list of the multi-owned assignment.

4.7.2 The Compilation System

Normally when compilation techniques are described, the data-parallel extensions are isolated from the rest of the compilation system into a separate phase, mostly because the compiler framework only allows a sequence of translations to the internal representation. In CoSy however different interaction patterns are possible between engines working on the same internal representation. Special attributes are needed to intersperse existing ‘classical’ optimizing engines with the owner-aware engines. As far as we know, there are no papers discussing this kind of problem.

4.7.3 Related compiler developments

Many different HPF compilation systems have been built, ranging from prototype HPF compilers to commercial HPF compilers (for HTML links to more information about these compilers see Appendix). Examples of prototype compilers are: Adaptor (GMD-SCAI), Annai (CSCS/NEC), EPPP (CRIM Montreal), Fortran-D (Rice University), FX Compiler (Carnegie Mellon University), VFC (University of Vienna), HPFC (Ecole des Mines de Paris), Paradigm (University of Illinois at Urbana-Campaign), and SHPF (University of Southampton and VCPC).

Examples of commercial compilers are DIGITAL Fortran (DEC), EXPERT HPF (ACE), HPF Mapper (EPC), HPFPlus (NAS), pghpf (PGI), VAST-HPF (PSR), xHPF (APR), XL HPF (IBM).
The majority of these compilers (both the commercial as well as the prototype versions) have been designed as a source to source translator. They transform HPF into either Fortran 77 or Fortran 90. The drawback of this design is that the HPF compiler will depend on the efficiency of the Fortran 77 or Fortran 90 compilers.

For both target languages (Fortran 77 and Fortran 90), optimal performance can not be reached. Fortran 77 compilers, being around for a long time are perfectly capable of doing advanced optimizations. However, most of the optimizations will fail to recognize the pattern they operate on because of the elaborate Fortran 77 output (due to the expressiveness of the HPF language compared to Fortran 77). When using Fortran 90 as target language the expressiveness is much better (array assignments are allowed), but the compiler themselves are not yet mature enough.

The PGI compiler was first designed for a shared memory machine and used one-sided communication. Later, an HPF compiler was released for a distributed memory machine which relied on a virtual shared memory machine. The extra overhead accompanied with virtual shared memory machines prevented good performance for a wider class of problems.

4.7.4 Alignment Analysis

A two-sweep algorithm comparable to the one mentioned in Section 4.4.2 has first been applied by Gilbert and Schreiber [30]. They have considered work distribution of array assignments on a fixed hardware topology (a SIMD like processor array). They show that an optimal work distribution can be found for a restricted class of problems (no common sub-expressions, only regular access functions, and known bounds) and a cost function with certain characteristics.

Chatterjee et al. [17] extended the work of Gilbert et al. [30] to array variables (user or compiler introduced), a larger set of array operations, and basic blocks as the scope of owner selection. Korstanje [45] removed some restrictions from Chatterjee’s work by allowing reordering of associative operators, the usage of irregular owners, and true multi-dimensional owner comparison. Our work goes even further by allowing owner generalization, owner privatization, and map application and extraction.

Our approach is more general in the sense that we provide support for the full HPF 2.0 language and allow optimizations with a local scope. The method integrates well with sequential optimizations and the distribution of work has the same two-level mapping as HPF data distributions. In addition, partially specified mapping information can be handled, ranging from ‘aligned to an inherited mapping’ to ‘aligned to a distributed template’. Therefore, the exact distribution onto processors does not need to be known at compile-time (e.g. inherited mappings).

The size of the search space in our approach is roughly the product of all owner options and is not related to the size of the processor array as in Chatterjee et al. [17]. Therefore, a full search algorithm can be applied in many more cases.
Most of the current HPF compilers only have a rudimentary align-engine. All work involved in data assignments are executed by the owner of the assigned data. This is called the owner computes rule (OCR). For some trivial cases, these compilers may deviate from the owner computes rule.

4.7.5 Basic Assignments

The automatic generation of message passing programs from data distribution specifications has been explored for some time in the context of various data parallel languages [13], [29], [62], [53], [43], [25]. The definition of HPF [32] has added some new data alignment and data distribution features for which no efficient solutions existed. Consequently, new results have been reported in [72], [18], [67], [68], [31], [37], [8], [3], [34] and more recently and concurrently with our paper [61], the block_cyclic distribution was further investigated in [71], [40], [39], [41].

Most of the above research concentrated on the efficient translation of array assignments, which can be viewed as their basic assignment. Restricting to array assignments avoids complicated cases where iteration spaces are non-rectangular, or the usage of multiple loop variables in access functions.

All communication needed inside an array assignment is handled completely by the run-time system. The communication occurs mostly from a mapped source array to a mapped destination array (for example the distributed array library dalib of the Adaptor HPF compiler (see Appendix)). This leaves no room to exploit the fact that the sender and receiver can compute parts of the basic assignment.

Because all communication is handled by the run-time system, the compiler can only detect different kinds of communication patterns that are supported by the run-time system. When such a communication pattern can not be detected the compiler has to fall back onto very inefficient element-wise communication.

Our basic assignment allows much more cases that can be handled efficiently. The support from the run-time system is brought back to the bare minimum of communicating tightly packed buffers.
5 Storage and Iteration Set Generation

Once a data mapping is given, we know exactly which element of an array is owned by which processor. However, the storage of all elements owned by a single processor still needs to be determined. When selecting a storage scheme it has to be efficient with respect to memory usage as well as to the costs of referencing the elements.

When the compiler needs to generate code for the packing or assignment phase (see Section 4.6.4) it needs to enumerate the local elements efficiently. Efficient in the sense that only a small overhead is allowed compared to the work inside the iteration space. Because in the packing and assignment phase local elements need to be referenced, storage and enumeration are closely linked to each other. Both topics, storage and enumeration, will be discussed below in more detail.

5.1 Storage

Efficient storage of mapped data is critical in two areas of our HPF compilation system. First the mapped data must be allocated efficiently, and secondly the send and receive buffers must be packed efficiently when communicating information. Both kinds of storage will be discussed in more detail in the next two sections.

5.1.1 Allocation of mapped arrays

When dealing with allocation of mapped arrays, two issues are important: The size of the local array size\textsubscript{local}, and the complexity of the function that calculates the local indices from the global indices, the so-called global-to-local function \( l = g2l(i) \).

The simplest way of allocating storage for mapped arrays, is to allocate the global size of the array on each processor node (i.e. size\textsubscript{local} = size\textsubscript{global}). This would make the global-to-local function very simple (i.e. \( l = g2l(i) = i \)). However, this would result in a parallel program that is not scaleable in memory, as the memory usage increases with the number of processors used to run the program.

Therefore, each processor should allocate only those array elements it actually owns. Allocation of some extra array elements, which may be never referenced (so-called memory holes), is allowed if this simplifies the global to local function.

As was shown in Figure 3, a block or cyclic distribution is just a special case of a block cyclic distribution, therefore we will only discuss the storage of a block cyclic distributed array. We will limit the description to the case of a one-dimensional mapped array, because storage for multidimensional arrays can be generalized straightforwardly. In Example 29 the most general HPF mapping is specified. No lower bounds and upper bounds are known at compile time. The same holds for the stride \( \alpha \) and offset \( \beta \) of the alignment and the block size \( m_0 \) and the cycle length \( m_1 \) of the distribution.
Example 29: The most general one-dimensional HPF mapping.

The corresponding template data layout is given in Figure 20. The template elements are distributed in blocks of size m in a round robin fashion. The first template block is aligned with the first processor element. The template elements are denoted as squares. The array elements are denoted as circles.

The relation between an array index i the template index t, and the row, processor, and column tuple (r, p, c) is given by the position equation as shown in Equation 1. \( m_t = m \times P_{\text{ext}} \) is the cycle length, i.e. the stride in template elements when the row counter is incremented by one. \( P_{\text{ext}} = (U_p - L_p + 1) \) is the extent of the corresponding processor dimension.
\[
\alpha \cdot (i - L_i) + \beta = m_i \cdot r + m_0 \cdot (p - L_p) + c \\
L_i \leq i \leq U_i \quad 0 \leq r < R_{ext} \\
L_p \leq p \leq U_p \quad 0 \leq c < C_{ext}
\]

Equation 1: The storage position equation and its bounding box

Notice that the variables \(i\) and \(p\) are running from their respective lower bounds up to their upper bounds, while \(r\) and \(c\) are running from zero up to, but not including their extents \((R_{ext} \text{ and } C_{ext})\).

Simply allocating the local template elements would still be a waste of memory, because many template elements do not correspond to array elements. The storage of the local elements would however become quite easy; just allocate locally a two-dimensional array \(A_{loc}\) of shape \((R_{ext} \text{ and } C_{ext})\), and the global-to-local function would be \((r, c) = (t \, \text{div} \, m_i, t \, \text{mod} \, m)\), where the template coordinate is \(t = \alpha \cdot (i - L_i) + \beta\).

A processor referencing its own local element corresponding with the global element \(A(i)\) has to calculate the template index \(t = \alpha \cdot (i - L_i) + \beta\). From this template index the row index \(r = t \, \text{div} \, m_i\), and the column index \(c = t \, \text{mod} \, m\) can be found by applying the template-to-local function \(t2l\). The \((r, c)\) tuple can be used as an array subscript \(A_{loc}(r, c)\)\(^{38}\) to reference the corresponding local element in \(A_{loc}\).

\(^{38}\)Whether this two-dimensional array is stored row-wise or column-wise is implementation dependent.
When zooming in on the local template of a processor we can recognize tiles of repeating array element patterns as depicted in the upper left part of Figure 21. The size of such a tile is \( \Delta r \) by \( \Delta c \) and contains only a single array element.

Because a tile can contain only one element, both rows and columns can be compressed, resulting in a tighter memory layout. Therefore, instead of referencing \( A_{kr}(r,c) \) we now use the compressed row and column indices \( r' = r/\Delta r \) and \( c' = c/\Delta c \) respectively, in \( A_{kr}(r',c') \) for the global-to-local index conversion. The global-to-local function \( (r',c') = g2l(i) \) can be interpreted as a function that returns the tile coordinates of the tile containing element \( i \).

Several of these tiles can be found each having a different size. We can distinguish three types of tile selection by looking for repeating patterns.

For **column-wise compression**, the elements occurring in the same row are compressed side by side. The compression factors are then \( (\Delta r, \Delta c) = (l, \min(\alpha, m)) \). The memory layout is shown in
the upper right part of Figure 21. Within a row, the global index of the leftmost element depends on the row used, and the global stride is a constant and independent of the row used.

For row-wise compression, the elements occurring in the same column are compressed side by side. The compression factors are \((\Delta r, \Delta c) = (\alpha / g, l)\), where \(g = \text{gcd}(m_l, \alpha)\), and \(\text{gcd}\) stands for the greatest common divisor. The memory layout is shown in the lower left part of Figure 21. Within a column, the global index of the topmost element depends on the column used, and the global stride is a constant and independent of the column used.

For tile-wise compression, the largest possible tile is found by putting an element in the upper left corner of a rectangular and drag the lower right corner as far as the rectangle contains one element. The compression factors are \((\Delta r, \Delta c) = (\alpha / g, g)\). The memory layout is shown in the lower right part of Figure 21. It can be seen as a row-wise compression followed by a column-wise compression.

When the order of the local elements must coincide with the order of the global elements (as mentioned in Chatterjee [18]), a permutation table can be used on the column index, i.e. \((r'', c'') = (r', \text{Tab}[c'])\). For example the table \(\text{Tab} = (1,0,2)\) would permute columns zero and one of the tile-wise compression in Figure 21, such that the elements per row will be in the same order as the global elements.

Tile-wise compression has the same characteristics as row-wise compression. Within a row, the global index of the leftmost element depends on the row used, and the global stride is a constant and independent of the row used. Tile-wise compression is the most efficient one, because its compresses with the largest compression factor.

Because, the block and cyclic distributions are special cases of the block_cyclic distribution, all the previous global to local conversions can still be used. Except that for block distributions the row-index \(r\) is always zero, and for the cyclic distribution the column-index \(c\) is always zero.

For a block or cyclic distribution, proper values for the coefficients of the position-equation can be found such that it can be treated as a block_cyclic distribution. Because of that, the compiler may assume a block_cyclic distribution for unknown (inherited) distributions (see Section 4.2.1).

### 5.1.2 Send and receive buffer management

One of the often-neglected aspects in designing a compilation framework for a data parallel language is the efficient use of send and receive buffer memory. Data-parallel programs are usually scaled up in problem size, such that they just fit into the available memory of the parallel machine. The memory needed for the send and receive buffer must not depend on the number of processors used. Otherwise, the program would not scale in memory.

In our framework, all elements destined for the same processor are packed side by side into the send buffer of that processor. All processor send buffers are packed side by side into one large send buffer. Simple bookkeeping while packing can store the starting address and length of the
send buffer for each processor. Notice that the partitioning of the send buffer into processor send buffers can (and in most cases will) be irregular.

The total size of all send-buffers on all processors is equal to the iteration volume of the associated basic assignment. The same holds for the receive buffers. Therefore, the extra memory usage per basic assignment is two times the iteration volume of the basic assignment. Because data-distributions can be quite unbalanced, care should be taken of overflowing individual processor memories, especially when duplicate values are packed.

5.2 Enumeration

Once a normalized volume assignment is attributed with one or two owners (see Section 4.6.4), the compiler must distribute the iteration space over the processors involved. The basic technique used for iteration distribution is called mask absorption. We first deal with enumerating the iterations of a one-dimensional normalized volume assignment with a single mask. Later, we show how to handle multi-dimensional normalized volume assignments with multiple masks.

5.2.1 Code generation

Suppose we have a one-dimensional iteration space and a three-dimensional owner with a linear alignment function \( a \times i + b \), as given in the top half of Example 30.

```c
    do { i = L:U:S | p \in P(*,block_cyclic(a*i+b),2) }
        ...
    od

    if p \in P(*,*,2))
        do i = L,U,S
            if p \in P(*,block_cyclic(a*i+b),*)
                ...
            fi
        od
    fi
```

Example 30: A naive owner implementation

The compiler must generate only those values of \( i \), for which \( p \) is an element of \( P(*,\ block_cyclic(a \times i + b), 2) \). A naive implementation could be used as given in the bottom half Example 30, where the mask expression is \( p \in P(*, \ block_cyclic(a \times i + b), *) \).

Although it will enumerate the local iteration set of processor \( p \) correctly, it will be very inefficient because the mask expression contains expensive modulo and divide operations that are evaluated each iteration. For most of the iterations the mask will evaluate into false (worst case a
fraction of \( 1 - 1/p_{\text{ex}} \), where \( p_{\text{ex}} \) is the extent of the distributed dimension). Therefore, the program does not scale in time when increasing the number of processors.

```c
// dtype = block or cyclic
if p ∈ P(*,*,2)
  (L',U',S') = lus1(dtype,L,U,S,a,b,map.dim[1],p[1])
  do i = L',U',S'
      ...
  od
fi

// dtype = block_cyclic
// order = row_wise or column_wise
if p ∈ P(*,*,2)
  (L',U',S') = lus21(order,L,U,S,a,b,p[1],...)
  do t = L',U',S'
      (L'',U'',S'') = lus22(t,p[1])
      do i = L'',U'',S''
          ...
      od
  od
fi
```

**Example 31: Mask absorption**

With mask absorption for a block or cyclic distribution, the mask expression is absorbed into the loop by reducing the loop bounds (also known as squeezing the iteration space). The function that does the loop bounds reduction, \texttt{lus1} is called the **squeeze function** (or **squeezer**) of the loop. It needs to know the original loop bounds \((L, U, S)\), the alignment stride \(a\) and offset \(b\) of the alignment function, and the processor index \(p\). It returns the reduced loop bounds \((L', U', S')\) (see top half of Example 31).

For a block_cyclic distribution, the original normalized volume assignment is transformed into a two-deep loopnest. The outer loop will enumerate the global indices of the starting points of the rows \((\text{order} = \text{row_wise})\) or the columns \((\text{order} = \text{column_wise})\), depending on the enumeration order as specified by \texttt{order}. The inner loop will enumerate the global indices given the starting point \(t\) (see bottom half of Example 31).

In all cases after absorbing one mask, corresponding with one dimension of the processor array, the iteration space at the inner loop is still a \(L:U:S\) triplet. That means that multiple masks can be absorbed one after the other, whether they come from another dimension of the same processor array or from another processor array. As long as the alignment function is linear in the same loop variable, the masks can be absorbed as shown above for that loop variable. Masks with non-linear alignment functions can not be absorbed. Some examples of the broad applicability of mask absorption are given in Example 32.
In the **first** loopnest, a non-rectangular iteration space is specified. The first dimension of the owner will squeeze the outer i-loop. The second dimension will squeeze the inner j-loop. The squeezer of the inner loop will have a function of the outer loop variable i as lower bound of the input triplet. This kind of squeezing is allowed because a parameterized squeezing method is used. The values of the parameters only need to be fixed for the scope of the corresponding loop.

Something similar occurs for the **second** loopnest, where we have a skewed owner. The squeezer of the inner loop will get an offset parameter which is a function of the outer loop variable i.

In the **third** loopnest, a diagonal of the processor array is specified. Two squeezer will one after the other, squeeze the same iteration space of loop variable i. The efficiency of applying two squeezer to such a loopnest is compared with other methods found in literature in Section 5.2.3.

The **last** loopnest can show up in the packing or assignment phase of a two-owned volume assignment (see Section 4.6.4).

When processor \( p \) is packing values for processor \( q \) in a certain order into a message, then processor \( q \) must read from the message of processor \( p \) in the same order. Notice that no information about the global index is packed and communicated. Therefore, the order of mask absorption in the packing and assignment phase is significant and must be equal.
5.2.2 Polyhedron scanning

Now that we know how mask absorption looks like in the generated code and how widely it can be applied, the question remains how to determine the enumeration of the owned elements of a processor dimension. The theory behind this enumeration technique is called polyhedron scanning. A polyhedron [65] is characterized by a set of linear diophantine equations (equations with integer-valued coefficients and variables) and a bounding box in which all solutions must lie.

\[
A \cdot (i - L_i) + B = m_1 \cdot r + m_0 \cdot (p - L_p) + c \\
L_i \leq i \leq U_i \quad 0 \leq r < R_{\text{est}} \\
L_p \leq p \leq U_p \quad 0 \leq c < C_{\text{est}}
\]

Equation 2: The enumeration equation and its bounding box

We will assume a block_cyclic distribution, because a block and cyclic distribution are special cases of a block_cyclic distribution.

For a block_cyclic distribution we have only one linear diophantine equation, the enumeration equation and four variables \((i, r, p, c)\) which are bounded (see Equation 2). Notice the strong relation with the storage position equation given in Equation 1. The only difference is in the definition of \(A, B, L_i\) and \(U_i\). The stride \(A\) and offset \(B\) are derived from a composition of the storage stride \(\alpha\) and offset \(\beta\) and the iteration stride \(S\), and offset \(L_i\). The bounds \(L_i\) and \(U_i\) are the lower bound an upper bound of the iteration space.

Because we only have one diophantine equation, we do not need the general approach of polyhedron scanning as described in [2]. We will use a simple technique of decomposing a diophantine equation with multiple variables into a sequence of diophantine equations with only two variables.

Let us focus on the variables \(i\) and \(r\) and suppose the other variables have a known value. After reordering of the coefficients of the enumeration equation in Equation 2 we get

\[
A \cdot i = m_1 \cdot r + g \cdot n_i, \\
g \cdot n_i = c + m_0 \cdot (p - L_p) - B
\]

Equation 3: The column-wise decomposed enumeration equation
We now have two linear diophantine equations each with only two variables \((i, r)\) and \((n, c)\), respectively. For these equations the solutions, indexed by \((j, j_2)\), are known and satisfy Equation 4, where 

\[
A \cdot i_0 = m_1 \cdot r_0 + g, \quad \text{and} \quad n_2 = m_0 \cdot (p - L_p) - B.
\]

\[
\begin{pmatrix}
L_i \\
0
\end{pmatrix} \leq 
\begin{pmatrix}
i \\
r
\end{pmatrix} \leq 
\begin{pmatrix}
i_0 \\
r_0 \\
1
\end{pmatrix} + 
\begin{pmatrix}
m_1 \\
A/g
\end{pmatrix} \cdot j_1 \\
\leq 
\begin{pmatrix}
U_i \\
R_{ext}
\end{pmatrix}
\]

\[
\begin{pmatrix}
n_1 \\
c
\end{pmatrix} \leq 
\begin{pmatrix}
m_1 \\
-1
\end{pmatrix} \cdot n_2 + 
\begin{pmatrix}
g \\
1
\end{pmatrix} \cdot j_2 \leq 
\begin{pmatrix}
C_{ext}
\end{pmatrix}
\]

Equation 4: The bounding box of \((j, j_2)\) for column-wise decomposition.

The values of \(i_0, r_0, g\) can be calculated by the Euclidean algorithm. Details of Diophantine equations and the Euclidean algorithm can be found in Rosen [63].

Now all solutions can be generated with two do loops. The outer loop generates all \(j_2\) values, which is bounded indirectly by the \(c\) interval. The inner loop generates all \(j_1\) values, which is bounded by the \(i\) interval and the \(r\) interval and indirectly by the value of \(n\). The fact that \(n\) is not bounded is denoted by a \(\ast\) in Equation 4.

Because the inner loop generates all solutions for a given column \(c\), this kind of enumeration is called column-wise enumeration. The solutions are generated in \((c, r)\) order.

If we first focused on the variables \(i\) and \(c\), we would have a row-wise decomposed position equation as given in Equation 5.

\[
A \cdot i = c + g \cdot n,
\]

\[
g \cdot n = m_1 \cdot r + m \cdot (p - L_p) - B
\]

Equation 5: The row-wise decomposed position equation

In this case the solutions can also be indexed by \((j, j_2)\) as shown in Equation 6.
\[
\begin{pmatrix}
L_i \\
0
\end{pmatrix} \leq \begin{pmatrix}
0 \\
-1
\end{pmatrix} \cdot n_1 + \begin{pmatrix}
1 \\
A
\end{pmatrix} \cdot j_1 \leq \begin{pmatrix}
1 \\
1
\end{pmatrix} \cdot n_2 + \begin{pmatrix}
m_1 \\
1
\end{pmatrix} \cdot j_2 \leq \begin{pmatrix}
1 \\
1
\end{pmatrix}
\]

Equation 6: The bounding box of \((i, j)\) for row-wise decomposition

Where \(n_2 = m^*(p - L_p) - B\). For row-wise decomposition no Euclidean algorithm is needed because the value of \(g\) is known beforehand, i.e. \(g = \gcd(I, A) = 1\).

Hiding the calculation of the loop bounds in squeezers results in an enumeration code as depicted in Example 31.

For masks containing a block distribution there will only be one row of solutions (i.e. \(r = 0\)), therefore the outer loop in row-wise enumeration does not need to be generated. For masks containing a cyclic distribution there will only be one column of solutions (i.e. \(c = 0\)), therefore the outer loop in column-wise enumeration does not need to be generated.

If the compiler does not know the distribution type it may assume a block \_cyclic\_ distribution type, and will generate a double loop instead of a single loop. The enumerated solutions however are the same, the enumeration itself would only be less efficient.

Notice that for column-wise/row-wise enumeration the loop variable \(i\), the row index \(r\), and column index \(c\) are **induction variables** of the inner loop. The same holds for the local index variable \(l = g2l(i)\). Therefore, the global-to-local function can be hoisted out of the inner loop as shown in Example 33.

```cpp
FORALL (i = ...) 
  A[i] = ...  

(L', U', S') = lus2l(L, U, S, ...);
DO (r = L':U':S') 
  (L", U", S") = lus2l(L', U', S', ...);
  l = g2l(L", ...);
  lext = g2l(L"*S"*, ...) - loff;
  DO (i = L":U":S") 
    A[i] = ...
    l += lext;
  OD
OE
```

Example 33: Hoisting out the global-to-local function.
5.2.3 Iteration set intersections

How does the iteration intersection as described above compare with work from other researchers? When iteration sets need to be intersected, three methods can be used: table intersection, numerical intersection, and symbolic intersection. Each method will be described below in more detail. The table intersection method has been used by Chatterjee et al. [18]. The numerical intersection method has been used by Benkner et al. [8] in the Vienna Fortran Compilation System (VFCS). The symbolic intersection method has been used in our CoSy compilation system.

Iteration sets need to be intersected when multiple masks need to be absorbed in the same iteration variable. This only occurs when send or receive-sets need to be generated, subscripts are coupled, or both. In this section, we will show the differences in the efficiency of generating iteration set intersections for a given example of coupled subscripts.

Each method is detailed below and illustrated with the example depicted in Figure 22. In the example, we have the array access function $A(i, i, i)$. The array $A$ is distributed $CYCLIC(2)$, $CYCLIC(3)$, $CYCLIC(4)$ on a $2 \times 2 \times 2$ processor arrangement.

![Diagram of intersection of local iterations of $A(i, i, i)$]

Figure 22: Intersection of local iterations of $A(i, i, i)$

Table intersection.

Table intersection uses tables as a storage format for iteration sets. In principle, these tables are as large as the iteration space itself. Luckily, a pattern can be found in the iteration set that repeats every cycle, the pattern cycle. Only this pattern needs to be stored into a table.

To find the pattern cyclic table for the local iteration set on processor $P[0, 0, 0]$, we first have to construct pattern tables $Pat$ for each individual dimension. We then have to find the start element,
i.e. the first value that all local iteration-sets of each dimension have in common. In Kennedy [41], it is outlined how this value can be found. Alternatively, applying the rowwise symbolic intersection method outlined below also yields the first element of the enumeration.

After having found the start value, each cycle of a dimension must be unrolled a number of times, such that the pattern cycle of all three dimensions can be found. In Figure 22, it can be seen that the pattern cycles in each dimension are 4, 6, and 8 respectively. Hence, the least common multiple is 24, implying that the cycles need to be unrolled 6, 4, and 3 times, respectively. For example the start element for processor $P[0, 0, 0]$ and $P[1, 1, 1]$ are 0 and 15, respectively.

From the starting elements on, the local iterations are enumerated per dimension to the length of the unrolled cycles. The derived local sets are intersected with each other to find common elements. The resulting table is used for enumerating the local iterations. In the example, it can be seen that the intersected pattern table for processor $P[0, 0, 0]$ is equal to $Pat = [0, 1, 8]$ and its cycle is 24 (denoted as a circle in the 3rd dimension in Figure 22). There is no closed expression for the number of elements in a cycle of an intersection table. This number is derived by counting the elements of $Pat$ during table intersection.

**Numerical intersection**

In the numerical intersection method, the format to store the iteration set is a list of regular sections, i.e. a list of lowerbound, upperbound and stride tuples. These regular sections can either describe the local elements per row (row-wise regular sections) or per column (column-wise regular sections).

To find the set intersections, we need to calculate per dimension all rowwise or columnwise regular sections resulting from the loops. All sections from the different dimensions are intersected with each other, resulting in a number of non-empty regular sections. For our example, this means a total of $2 \times 3 \times 4 = 24$ intersections, if we apply them columnwise. Columnwise intersection is bound by the blocksize $m$. In general, the numerical intersection complexity is $\Pi m$, where $m$ are the blocksizes of the dimensions to be intersected. Rowwise numerical intersection has no such limit. The number of rows to be intersected is dependent on the maximum column length in each dimension (in our example $9 \times 6 \times 5 = 270$ intersections). Hence, rowwise numerical intersection might get very expensive\(^\text{59}\).

**Symbolic intersection**

Symbolic intersection does not use a format to store iteration sets. It uses a loopnest structure to enumerate the local elements directly. Symbolic intersection can be done by applying the rowwise or columnwise local enumeration methods in a repeated way.

\(^{59}\) In this example, column-wise intersection is to be preferred. Other examples might prefer a row-wise intersection, depending on the number of rows and columns of the layout of the local elements.
This works as follows. Take columnwise enumeration. The loop resulting from the first dimension can be viewed as a partial local-iteration sequence itself. We can use this partial local-iteration sequence to span the columnwise iteration for the second dimension. Subsequently, this process can be repeated for the third dimension (leading to four nested loops). This process is visualized in Figure 22. The first column of the first dimension spans two columns in the second dimension, each spanning a single column in the third dimension, etc. In total, the columns of regular sections are generated in this example.

Symbolic intersection has the advantage that no storage\(^{40}\) is needed while enumerating the elements. The regular structure of enumeration is reflected in additional loops enumerating either the rows or columns of the distributed dimensions. These additional loops also facilitate the introduction of induction variables, which can hold partial results of the global to local functions as shown in Section 5.2.2.

Because the intersection is symbolic, no restrictions are needed on the loop bounds. Inner loops may have bounds depending on outer loops, allowing for example triangular iteration spaces.

### 5.3 Related work

#### 5.3.1 Storage

Enumeration of local array elements and the efficient storage and access of distributed arrays are two intertwined problems that are mostly addressed as one problem. Chatterjee et al. [17] use a local storage compression scheme in which array elements are stored in lexicographical order without holes. Because the Finite State Machine they use for enumerating elements directly enumerates local elements, explicit global-to-local calculations are not necessary.

In Kaushik et al. [39], similar dense storage compression schemes are employed. Each scheme is specific for an enumeration method. Changing the enumeration method for an array leads to a reallocation of the array. In addition, no global-to-local formula is given; resulting in table-generation time overhead for translating the global index sets to the local index sets.

Stichnoth et al. [68] and Midkiff [50] use a block compression method with the cycle number as a second index. Ancourt et al. [3] derive a formula for a columnwise compression method. Their global-to-local function remains rather complicated, although they do remark that in many cases solutions that are more efficient can be obtained.

Benkner [8] describes a columnwise solution with dense storage and outer loop tables. His global-to-local function is more complicated than the ones described in this thesis.

---

\(^{40}\) More important no estimate of storage size is needed. For the table and numerical intersection methods, estimating the storage size can be so complex or even impossible, resulting in severe restrictions to the allowed intersections.
A run-time technique is proposed by Maheo and Pazat [47], to avoid complex global-to-local functions. Their method is based on page driven array management. The main advantage of their approach is that they can provide a very efficient global-to-local function. This comes at the expense of storage overhead and page management.

5.3.2 Enumeration

Various authors have addressed the problem of local set enumeration for canonical data distributions in data parallel languages, such as block, cyclic and block_cyclic distributions. Early optimizations only considered direct distributions on arrays without the complex alignment functionality as provided in HPF.

The first optimizations were reported by Callahan and Kennedy [13] and Gerndt [29]. They considered non-aligned block distributions with linear array access functions. Gerndt also showed how overlap could be handled. In Paalvast et al. [53] a solution for monotone array access functions and block distributions was given.

Solutions for cyclic and linear array access functions have been independently reported by Koelbel and Mehrotra [43] and Koelbel [42] for the language Kali, and Paalvast et al. [53] for the language Booster [54], [55].

For block_cyclic distributions with no alignment, a rowwise and columnwise solution has been given in Paalvast et al. [56], [55]. The rowwise solution allows array access functions to be monotone and the columnwise solution requires the array access function to be linear.

Later publications also considered alignment. Most results assume that dimensions are independent of each other and/or have restrictions on the class of alignment functions allowed. Tseng [72] considers block and cyclic distributions with an alignment stride of one.

Two related approaches have been presented by Stichnoth et al. [67], [68] and Gupta et al. [31]. Essential in these approaches is the notion of virtual processors for solving the block cyclic case. Each block or cyclic solution for a regular section of an array is assigned to a virtual processor, yielding a so-called virtual-block or virtual-cyclic view, respectively. Stichnoth et al. [67], [68] use a virtual-cyclic view and use the intersection of array slices for communication generation.

Gupta et al. [31] present closed form solutions for non-aligned arrays for both the virtual-block and virtual-cyclic view. This method is extended to aligned arrays in Kaushik et al. [39]. To reduce calculation overhead in the virtual processor approach, Kaushik et al. [37] propose a method to reuse the calculations for classes of related access functions.

The virtual processor methods are related to the rowwise and columnwise methods outlined in this thesis. Each template column or, alternatively, each template row can be considered as a virtual processor. For each virtual processor, a regular section is determined. Some of the template rows or columns may have empty regular sections. Virtual processors are subsequently assigned
to real processors. The virtual processor method requires substantial bookkeeping and is not fully parameterized.

Chatterjee et al. [18] describe a non-linear method to enumerate local elements by using a finite state machine. They observed that the sequence of positions of elements in the rows of a template shows a pattern that repeats itself after a number of rows. Let us call such a range of rows a pattern cycle of a distribution, and let us call this enumeration method pattern cyclic enumeration. Chatterjee et al.'s contribution is the construction of an algorithm to find this pattern and place the elements in an in-order sequence. From this sequence, a finite-state machine FSM is constructed which is used to successively access each element.

In the original paper of Chatterjee et al., the construction of the FSM requires a full sorting operation. Recent papers describe more efficient methods [34], [71], [40], [41]. In [34], a linear algorithm for constructing the FSM for two special cases is given. Linear algorithms for the general case are given in [71], [40], [41]. Kennedy et al. also showed [41] that their method could be used without a table, using a demand driven evaluation scheme, at the expense of some performance.

As shown in this thesis, rowwise and columnwise enumeration tends to be complementary: One is inefficient where the other is efficient and vice versa. The efficiency of pattern cyclic enumeration is related to rowwise enumeration. In effect, pattern cyclic enumeration allows one or more rows to be enumerated in the inner loop.

How many rows are enumerated is strongly dependent on the alignment and index access function applied. This dependency can only be relaxed by enumerating more than one pattern cycle at once, at the expense of applying a two-level table scheme as shown in [41] or at the expense of larger tables (by unrolling the table).

Ancourt et al. [3] use linear algebra techniques to numerically construct enumeration and communication sets for full HPF mappings. They also present a symbolic solution for columnwise local index enumeration, with the restriction that the array access must be dimension wise independent.

Linear algebra techniques can be quite costly when not all parameter values are known at compile-time. This might be a problem in HPF programs. For example, the number of processors is often only known at run-time. To avoid this, Ancourt et al. [3] also present a symbolic solution for columnwise local index enumeration.

In Germain et al. [28] the symbolic method is extended for cyclic distributions and array accesses that are affine functions of the loop variables. Midkiff [50] presents symbolic solutions for rowwise enumeration and set intersection. Le Fur et al. [25] give solutions for commutative loop nests on block distributed and non-aligned arrays.

In Reeuwijk et al. [61], symbolic solutions are presented for rowwise and columnwise enumeration for block_cyclic distributions and array accesses that are affine functions of the loop vari-
\*RELATED WORK

ables. This technique is further developed and integrated in a compilation framework in this thesis.

In Wang et al. [74], the various methods described in Refs. [67], [31], [50], [18] and [71] are compared in terms of performance for array statements with no alignment to templates (no storage compression) and compile-time knowledge of distribution parameters.

All methods in their study use numerical techniques (tables or numerical set intersection) except for the rowwise method of Ref. [71] (this last method is only used in comparing local iteration set performance). They show that for this class of HPF array mappings, good speed-ups are achieved.

Sips et al. [66] show that efficient set enumeration and local storage schemes can be derived for the general case. Moreover, if a local-storage compression scheme is collapsible (i.e. the local row or column index can be compressed by index division), it can efficiently be combined with any set enumeration scheme.
6 Optimizations

The optimizations that are possible within the compilation framework can be divided into two classes: pre-SPMD optimizations and post-SPMD optimizations. Each optimization is designed to only have a local effect. This allows them to be used in different schemes. Like one after the other, mixed, or repeatedly applied until no effect can be observed. It guarantees an optimization sequence that best suits the given context, without having to rely on complex patterns that have to be found to trigger a certain optimization. The next sections give examples of optimizations that are possible within the framework.

6.1 Pre-SPMD optimizations

Pre-SPMD optimizations have an effect on the data and work distributions as a whole, and thereby have an effect on the performance that is related to communication between processors. Pre-SPMD optimizations take place before the SPMD engine has lowered the internal representation. Because Pre-SPMD optimizations have a notion of a global address space like HPF, they can usually be expressed at the input language level. For those cases where the optimization can not be expressed at source level, the input language should be extended (see Section 6.3 about HPF language extensions).

6.1.1 Message merging

The SPMD engine can translate a two-owned assignment into an efficient SPMD program, using the techniques described in Section 4.6. However, the applied technique allows a much broader range of input that can be handled. Let us look at the independent forall as given in Example 34.
// independent
forall (I = ... on P_{rec}(I), P_{send}(I))
    = ... on P_{send}(I){expr1} =
    = ... on P_{send}(I){expr2} =
end forall

typedef struct
{
    typeof(expr1) Fld1;
    typeof(expr2) Fld2;
} Tbuff

Tbuff *sbuff, *rbuff;

do ...
    sbuff[cnt].Fld1 = expr1;
    sbuff[cnt].Fld2 = expr2;
    cnt++
  od

  ...

do ...
    = ... rbuff[cnt].Fld1 =
    = ... rbuff[cnt].Fld2 =
    cnt++
  od

Example 34: Merging messages

The sequence of statements in the body of the forall forms a basic block; i.e. all control flow enters at the beginning of the block and will only leave at the end of the block. The basic block is a two-owned basic block. For each iteration I, the complete basic block can be evaluated at P_{rec}(I) except for the two expressions expr1 and expr2 that are owned by P_{send}(I).

The data flow between expr1 and its parent node is flowing from P_{send}(I) to P_{rec}(I) and is equal to the data flow of expr2 and its parent node. The evaluation of expr1 can not have a side effect on the value of expr2, because in CoSy all expressions are side effect free. The first assignment can not have a side effect on the value of expr2. Because, if it did then expr2 would need to use a value that is assigned on the lhs of the first assignment, which is owned by P_{rec}(I). But that would mean that expr2 can not be single owned by P_{send}(I).

Therefore, the two expressions can be evaluated and aggregated into a single message. The compiler can construct a new structure type Tbuff based on the types of the two expressions. The packing becomes a list of assignments to each field of Tbuff, and an increment of the buffer counter cnt. At the original places of the expressions proper references to the buffer fields are inserted. Data dependencies between the statements at the receiving side P_{rec}(I) are still observed in the assignment phase.
This optimization can be particularly helpful for legacy Fortran 77 code that is migrated to HPF, without altering the data structure. Because Fortran 77 does not have derived types (or C type structures), users have to revert to a set of arrays and assign each array individually. As this style of programming occurs frequently in applications written in Fortran 77, it will also occur frequently in the migrated HPF programs. Therefore, it is quite likely that the applicability of this optimization will be high.

### 6.1.2 Hoisting assignments

One of the most effective optimizations in vector computers is scalar expansion. A scalar (or array) assignment inside a loop is hoisted out of the loop by adding an extra dimension to the type of the scalar (or array). A similar optimization can be used for mapped scalars (or arrays) as shown in Example 35.

The two-owned scalar assignment can be hoisted out of a loop by adding an additional dimension to the lhs variable's type. The extent of this dimension is equal to the extent of the loop (i.e. the loop count). The mapping for this new dimension of the lhs variable (mapped \((i)\) on \(P_{\text{rec}}(i)\)) can be derived by adding the loop variable \(i\) to the parameter list of the scalar mapping (mapped \(\) on \(P_{\text{rec}}(i)\)). Owner generalization (see Section 4.3.3) might be needed on the resulting mapping to enforce a mappable owner.

```fortran
do i = expr1, expr2
  T x : mapped () on P_{\text{rec}}(i)
  on P_{\text{rec}}(x = \_ on P_{\text{send}}(expr) \_)
  \_ x \_
od

T x[expr1:expr2]: mapped (i) on P(i)
forall (i = expr1:expr2 on P_{\text{rec}}(I), P_{\text{send}}(i))
  x(i) = \_ on P_{\text{send}}(I)(expr) \_
end forall
do i = expr1, expr2
  \_ x(i) \_
od
```

Example 35: Hoisting an assignment

The analysis whether to hoist the assignment out of the loop is the same as for non-mapped data. The fact that parts of the code are mapped does not interfere with the hoisting analysis. The impact of this kind of optimization is significant because it aggregates a sequence of small messages into a single large one, and thereby avoids the large communication startup times per message.
The hoisted assignment can float upwards in the control flow and add additional dimensions when crossing do loops, as long as data-dependencies allow it.

6.1.3 Squeezing independent do loops

For independent loops (i.e., loops without loop carried dependencies) with a single owned body the owner mask can be absorbed into the iteration space of the do loop (see Example 36). This is exactly the same kind of optimization when masks are absorbed in the iteration space of a normalized volume assignment. Mask absorption is just more widely applicable than just to forall loops. It only applies to independent do loops because mask absorption alters the order of iterations.

```
// independent
do i = ...
on p(i)
  ...
end
```

```
// independent
do { i = ... | p(θ) ∈ p(i) }
```

Example 36: Mask absorption for independent loops

6.1.4 Stencil references

An important class of optimizations can be applied when dealing with stencil references between the owners of a two-owned assignment. A two-owned stencil assignment has owners, which differ by a constant (named the stencil offset) in their alignment functions. The compilation system is capable of detecting such stencil assignments. Stencil assignments can be lowered more efficiently than two-owned assignments.

For example, let array $A$ be mapped $(i)$ onto $P(block_cyclic(a \cdot i + b, m_0, m_1))$, then for all the local indices of $A(i)$ on processor $p$, the local indices of $A(i + i_off)$ are either owned by $p_1' = (p + p_off) \mod m_1$ or $p_2' = (p + p_off + Δp) \mod m_1$, where $a \cdot i_off$ corresponds to $(r_off, p_off, c_off)$ and $Δp = \{+1, 0, -1\}$. The position equation $t = a \cdot i + b = m_1 \cdot r + m_0 \cdot p + c$ associates a template index $t$ with a $(r, p, c)$ triplet. This triplet can be seen as a unique three-digit number with weight $(m_1, m_0, l)$, respectively. Because $t' = a \cdot (i + i_off) + b = (a \cdot i + b) + a \cdot i_off = t + a \cdot i_off$ the same equation must hold between their three-digit representations, i.e., $(r', p', c') = (r, p, c) + (r_off, p_off, c_off)$. The processor that can be reached $p'$, can be expressed as $p' = (p + p_off + Δp) \mod m_1$, where $Δp = (c +
\(c_{\text{off}} \div m_0\) represents a single carry (+1), borrow (-1) or neither case (0) from the last digit \(c\) to the second digit \(p\). Because \(0 \leq c < m_0\), \(c + c_{\text{off}} < 2m_0\). Therefore, there can only occur a single carry or borrow. Of course when \(c_{\text{off}} = 0\), only one processor can be reached.

The scanning of the processor array dimension in the packing and assignment phase can be reduced to maximally two values \(p_1\) and \(p_2\), as was also noted by Chatterjee [18].

When the stencil offset \(a \cdot i_{\text{off}}\) between two references is less than the blocksize \(m_0\), i.e. \((r_{\text{off}}, p_{\text{off}}, c_{\text{off}}) = (0, 0, c_{\text{off}})\), and the assignment has equal lhs and rhs expressions, then the stencil assignment is called a two-owned neighboring stencil reference. In that case, the compilation system can generate a neighboring stencil assignment as shown in Example 37. A neighboring stencil assignment has equal array references as lhs and rhs expressions. The sending processor and the receiving processor form a neighboring stencil reference. This neighboring stencil assignment will update the shadow area of \(B\) on the receiving processor with values of \(B\) owned by the sending processor.

```plaintext
A,B: mapped (i) on P(block_cyclic(i))
B: shadow(1:1)

forall (i = _ on P(block_cyclic(i)),P(block_cyclic(i+1)))
A(i) = A(i) + on P(block_cyclic(i+1))(B(i+1))
end forall

// annotated as neighboring stencil assignment
forall (i = _ on P(block_cyclic(i)),P(block_cyclic(i+1)))
B(i+1) = on P(block_cyclic(i+1))(B(i+1))
end forall
forall (i = _ on P(block_cyclic(i)))
A(i) = A(i) + B(i+1)
end forall
```

Example 37: A stencil optimization

For a neighboring stencil assignment an important stencil optimization (as first described by Gernd [29]) can be applied as shown in Example 37. It exploits the fact that when enumerating the local elements of \(A(i)\) on a processor \(p\) almost all elements of \(B(i+1)\) are locally available, except the elements \(B(i+1)\) for which the owner of \(A(i)\) has a corresponding \((r, p, c)\) tuple where the column \(c\) is equal to \(C_{\text{off}} \cdot l\). For those values of \(A(i)\), the neighboring processor \((p+1) \mod m_1\) owns the element of \(B(i+1)\).
Using an HPF SHADOW directive allows the local column extent \( C_{ext} \) to be extended at both ends with a shadow width. The shadow area (shown in Figure 23 as the gray area) that will also be allocated can be used to store remote values of \( B \). Notice that a shadow specification is also, like a mapping specification, a type extension. Explicit copying is needed between equally typed arrays with different shadow widths.

With the proper shadow width in place, a two-owned neighboring stencil assignment can be used to pack and send the elements \( B(i+1) \) at owner \( P(block\_cyclic(i+1)) \), receive them at owner \( P(block\_cyclic(i)) \), and unpack them into \( B(i+1) \), i.e. in the shadow area of \( B \) as shown in Example 37. The shadow area of \( B \) at owner \( P(block\_cyclic(i)) \) now contains fresh local copies of the remote values.

Because, there is no difference in referencing the local area or the shadow area of \( B \), while enumerating the local iteration space, the final assignment in Example 37 is a single-owned assignment.

Adding shadow areas can increase the array size considerably. With a blocksize \( m \) and a shadow of \([lo:hi]\) the array size is roughly increased by a factor of \( f = (m + lo + hi)/m \). Especially for cyclic distributions the increase is at least a factor of two. However, for cyclic distributions, the whole stencil optimization hardly makes sense, because there are no locally available elements that can be reused.

### 6.1.5 Reductions

Important classes of assignments are assignments containing a reduction operator or reductions. In a reducing assignment, the iteration space is larger than the number of elements that is updated.
To allow a sum reduction as given in Example 38, the semantics of the forall needs to be extended. For the extended forall, the lhs expression does not have to evaluate to a different array element for each iteration in the iteration space. The order in which the iterations are processed may not influence the final values of the assigned array elements.

The complete iteration space is generated by the dimensions \( \mathbf{I}, \mathbf{J} \). \( \mathbf{I} \) denotes the non-reduced dimensions, \( \mathbf{J} \) denotes the reduced dimensions. The reduction operation itself must be a commutative operation (like +, *, min, and max).

```
forall ((I,J) = ... on P_{recv}(I), P_{send}(I,J))
  A(I) = A(I) + on P_{send}(I,J) \{ expr(I,J) \}
end forall
```

\[ \Rightarrow \]

```
t(I,p): mapped (I,p) on P_{send}(I,p)
forall ((I,J) = ... on P_{send}(I,J), P_{send}(I,J)) //intra-processor reduction
t(I,\theta) = t(I,\theta) + on P_{send}(I,J) \{ expr(I,J) \}
end forall
forall ((I,p) = ... on P_{recv}(I), P_{send}(I,p)) //inter-processor reduction
  A(I) = A(I) + on P_{send}(I,p) \{ t(I,p) \}
end forall
```

Example 38: A reduction assignment

The idea is to split the dimensions of \( \mathbf{J} \) into an intra-processor reduction followed by an inter-processor reduction. The intra-processor reduction is a local operation; the inter-processor reduction requires communication. The communicated volume is however much smaller than the iteration space. The general sum reduction as is given in Example 38, will be transformed into a sequence of two foralls.

The first forall is an \textit{intra-processor reduction} because of the special mapping of \( t \). The second dimension of \( t \) is distributed \textit{block}(I), i.e. the global index \( p \) corresponds directly to the processor index. By using the special index specifier @ the complete reduction over the \( \mathbf{J} \) dimensions can be stored locally into \( t(I,p) \) for each processor involved. The compiler can easily verify that \( P_{send}(I, \@ ) \subseteq P_{send}(I, \mathbf{J}) \) and therefore the complete assignment is a single-owned assignment that can be evaluated locally.

The second forall is an \textit{inter-processor reduction} because it accumulates for a given \( I \) all partial reduction results of processors \( P_{send}(I, p) \) onto a single processor \( P_{recv}(I) \). This forall shows the advantage of allowing calculations in the assignment phase.
6.1.6 Masked assignments

So far, we have not discussed a two-owned assignment with a user-specified mask expression, \textit{mask}, depending on the iteration space \( I \), as shown in Example 39. The fact that the mask expression can also be distributed over a processor array is denoted with an extra owner attribute \( P_{\text{mask}}(I) \). Several relations between the owner of the mask and the sending and receiving owners can occur. Each of these relations will be discussed in more detail below. The compiler will try to verify each of these relations in the order given below.

\begin{verbatim}
forall (I = ...; mask(I) on P_{recv}(I), P_{send}(I), P_{mask}(I))
    ... = ...
end forall
\end{verbatim}

Example 39: A volume assignment with a mask

If the mask can be evaluated on both owners (i.e. \( P_{send}(I) \subseteq P_{\text{mask}}(I) \land P_{recv}(I) \subseteq P_{\text{mask}}(I) \)), then the mask can be used both in the packing phase as well as in the assignment phase.

![Buffer layout with mask piggybacking.](image)

Figure 24: Buffer layout with mask piggybacking.

Else, if the mask can only be evaluated at the sending owner (i.e. \( P_{send}(I) \subseteq P_{\text{mask}}(I) \)), then the sending processor packs the mask values. When the mask value evaluates to \textit{true} the rhs value is also packed into the send buffer. The receiver unpacks the mask from the receive buffer. When the mask value evaluates to \textit{true}, it uses the rhs value to evaluate the complete expression and assigns its value to the lhs address (as shown in Figure 24).
forall (i = ..., b(i) != 0 on ..., ...,)
    a(i) - a(i) + on ... (1/b(i))
end forall

...

mask = b(i) != 0;
*buff_send = mask;
buff_send += sizeof(boolean); // mask packing phase
if mask
    *buff_send = 1/b(i);
    buff_send += sizeof(1/b(i)); // value packing phase
fi
...

mask = *buff_recv;
buff_recv += sizeof(boolean);
if mask
    value = *buff_recv;
    buff_recv += sizeof(1/b(i)); // value unpacking phase
    a(i) - a(i) + *buff;
fi

Example 40: Lowering of masked assignments

In Example 40 the main packing and unpacking details of the SPMD generated code are given. The sizeof function that is used in the example returns a compile-time known integer value, equal to the size of the type used, or the size of the value the expression will evaluate to. When sizeof is applied to an expression, the type of the expression is analyzed at compile-time to determine its size; the expression itself is not evaluated. The complete process of sending mask information along with the rhs values is called **mask piggybacking**.

When the previous two relations can not be found, a temporary variable is introduced which is mapped onto the generalized owner of the rhs, and the previous two relations are checked again, as shown in Example 41.
6.1.7 Semi-regular assignments

A semi-regular assignment is a two-owned assignment, where one of the owners is mappable (regular) and the other is non-mappable (irregular). When the receiving owner is non-mappable, it is called a scatter assignment. When the sending owner is non-mappable, it is called a gather assignment.

When both owners are non-mappable, it is called an irregular assignment. For an irregular assignment the compiler can introduce a temporary variable mapped onto the generalization of one of the two owners, or select a new mapping. The irregular assignment can then be transformed into a sequence of a gather assignment followed by a scatter assignment as shown in Example 42.
Because gather- and scatter assignments have at least one regular owner, the communication is relatively efficient; i.e. all owner masks of the regular owner can be absorbed in the packing and assignment phase.

### 6.1.8 Inspector/Executor

When a communication pattern does not change between loop invocations, some compilers will use the inspector/executor paradigm, as described by Sussmann et al [69]. The schedule `sched` containing all the necessary bookkeeping is initialized in a separate phase called the inspector phase. The actual communication and evaluation occurs in the executor phase. Because the schedule does not change between loop invocations, it can be hoisted out as shown in Example 43.

```
do t = ...
    ...
    forall (I = _ on P_{recv}(I), P_{send}(I))
    _ = _
    end forall
end

⇒

Schedule sched
// inspector
sched = schedule (I = _) between P_{recv}(I) and P_{send}(I)

do t = ...
    // executor phase
    forall (I =_ scheduled sched)
    _ = _
    end forall
end
```

Example 43: Hoisting out expensive schedules

The inspector/executor paradigm is used when the inspector phase is very expensive compared to the executor. The use of this kind of schedule hoisting is restricted and its applicability must be asserted by the user. For instance, it is the user’s responsibility to assert that the communication pattern has indeed not changed between the definition (initialization) of the schedule, and its use.

Our framework is very flexible with respect to irregular assignments. The generation of the packing and assignment phases for semi-irregular assignments is so efficient\(^{41}\), that we expect that it will outperform the executor phase. Which method is best, strongly depends on how many times the same schedule can be reused. The exact turning point is currently not known. When

\(^{41}\) All loop variables are candidates to be put in processor registers and the only information stream from the relatively slow main memory to the processor is from the data itself, not from some extra schedule data.
schedules can moderately be reused, semi-regular assignments will outperform the inspec-
tor/executor approach.

6.1.9 Global-to-Local offset reuse

When an array is referenced twice or more inside the same scope of an owner, the global-to-local function can be reused by applying common sub-expression elimination. For example, in Example 44, an array, which is mapped in its first dimension, is accessed twice in the same expres-
sion. Because expressions are side effect free in our compilation system both values of the
global index \( i \) will be the same, i.e. they form common sub-expressions. Because both will be
replaced with a local index \( g2l(i, \ldots) \) once the SPMD engine has transformed the code, the value of
\( g2l(i, \ldots) \) will also be a common sub-expression. The compiler is free to store the value of the
local index \( g2l(i, \ldots) \) into a variable \( l \), and use it twice in the transformed expression. The reuse
can be exploited, regardless of the regularity/irregularity of the access functions.

```c
int A[N][M]: mapped (i) on P[block[a*i+b],*)
  A[i][j] = A[i][k]
  l = g2l(i,\ldots);
```

Example 44: Scalar global-to-local offset reuse

The same holds for common global-to-local functions inside a normalized volume assignment as
shown in Example 45.

```c
forall (i = \ldots)
  A[i][j] = A[i][k]
end forall

L',U',S') = lus1(L,U,S,\ldots);
\( l = g2l(L,\ldots); \)
\( l_{\text{off}} = g2l(L+U,S,\ldots) - l; \)
do (i = L':U':S')
  l += l_{\text{off}};
```

Example 45: Global to local offset reuse.
Global-to-local offset reuse is best implemented as a pre-SPMD optimization. When implementing it as a post-SPMD optimization, the compiler must recognize that several local strides and offsets are initialized to the same value and can in fact be reduced to just one local stride and one local offset, which is too complicated for most compilers.

When implementing it as a pre-SPMD optimization, the optimization simply reduces to common sub-expression elimination inside a single owner scope. The temporary variable used to store the common value is a private variable inside that owner scope. A private variable is allocated only by the active processor set of that scope. Although several variables with the same name exist at the active processor set, none of them needs to have the same value. The values are private to the processor on which they reside.

6.2 Post-SPMD optimizations

Post-SPMD optimizations take place after the SPMD engine and effect the computation performance of the generated code per processor. None of these optimizations will alter the communication patterns between the processors. The optimizations are ‘classical’ sequential optimizations, like common sub-expression elimination, copy propagation, dead-code elimination, and constant folding.

The difference with ‘classical’ sequential optimizations occurs in the special treatments of the SPMD introduced compiler known functions like squeezers, and global-to-local and global-to-processor functions.

Because the SPMD-introduced functions are compiler known functions (i.e. known to be computationally expensive and largely occurring inside loops), priority can be given to them when analyzing code. These could for example be the first candidates for invariant code motion.

6.3 Related work

Most of the optimizations found in the literature focus on just one or two aspects of the compilation system. For instance, as mentioned earlier Chatterjee et al. [18] use a local storage compression scheme in which array elements are stored in lexicographical order without holes. This may improve the memory usage but it sacrifices the flexibility to mix different optimizations.

The flexibility to select row-wise or columnwise storage and independently row-wise or columnwise enumeration can greatly reduce the enumeration overhead as is shown in Section 8.2.1. The fact that enumeration is done symbolically allows the absorption of different dimensions of owners in each loop of the iteration space.

Different approaches are used in the literature to handle irregular assignments. The Inspector/Executor approach introduced by Sussman et al. [69] relies on the PARTI run-time system handling all the details of the actual communication. However, soon became clear that the compiler needed some more hints on where to define the schedule and where it can be reused.
Brezany et. al. describe in [11] a further optimized PARTI run-time system that avoids sending duplicates.

Brandes [10] uses the TRACE directive to inform the compiler which arrays need to be traced. When those arrays are used as indirection arrays the run-time system can easily determine whether the array has changed since the last schedule calculation.

Benkner [8] introduced the HALO directive to allow a larger class of mappings for which schedules can be reused. The concept of halos allows the user to provide to the compiler, explicit information about non-local data access-patterns in addition to the distribution of arrays.

Halos can be considered as a generalization of HPF-2 shadows to irregular, possibly dynamically changing, access-patterns. In HPF+ [15], high-level communication primitives for halos are provided to enable the programmer to avoid redundant communication, to aggregate communication, and to overlap communication with computation.
7 HPF Extensions and restrictions

Given the fact that the above compiler model allows much more freedom than the HPF language, it is a natural wish to extend the language to exploit the full potential of the model. On the other hand, the HPF language needs to be restricted in those cases where expressiveness is not compromised and the possibility for powerful analysis is greatly enhanced.

The idea is to extend the language such that intermediate representations resulting from internal transformations can also be expressed in the source language. It allows the user to express algorithms in more detail without having to rely on an intelligent compiler. It also allows intelligent source to source translators, which output can be compiled afterwards with a less intelligent compiler.

In the next subsections different language extensions are given together with their motivating examples.

7.1 Parallel assignment

As shown in Example 34 several messages can be merged when the sending and receiving processors coincide. Without relying on a compiler clever enough to find out that messages can be merged, it can easily be specified when parallel assignments are allowed as shown in Example 46.

```
(lhs1, ..., lhs_n) = (rhs1, ..., rhs_n)

FORALL (...) 
  (lhs1, ..., lhs_n) = (rhs1, ..., rhs_n)
END FORALL
```

Example 46: Parallel assignment

A parallel assignment consists of a rhs tuple and a lhs tuple. Both tuples have the same size. The expressions in the rhs tuple are evaluated in arbitrary order before they are assigned in arbitrary order to the locations in the lhs tuple. The parallel assignment can always be lowered onto a sequence of volume assignments with the same volume42. A parallel assignment may occur at every place where a normal assignment is allowed, particularly inside a forall statement.

When the compiler analyses the parallel assignment, it can partition the original multi-owned assignment into a sequence of parallel assignments for which each tuple has only one owner. The resulting parallel assignments are then either single-owned or two-owned assignments.

---

42 Several temporary arrays might be needed to avoid introducing loop-carried dependencies.
By allowing parallel assignments, legacy FORTRAN 77 programs which only use scalars and arrays can easily be parallelized without introducing new types or retyping the used variables.

```fortran
!HPFS INDEPENDENT
FORALL (I = 1:N)
  AX(I) = EXPR1;
  AY(I) = EXPR2;
  AZ(I) = EXPR3;
END FORALL

⇒
FORALL (I = 1:N)
  (AX(I),AY(I),AZ(I)) = (EXPR1, EXPR2, EXPR3)
END FORALL
```

Example 47: Example of a parallel assignment

The first part of Example 47 shows a FORTRAN 77 code fragment where AX, AY, AZ form actually the three components (x, y, z) of an array A of vectors. In the first part, the compiler has to check several data dependencies between the assignments as described in Section 6.1.1. For the second part, the compiler only has to check whether the assignment is two-owned.

### 7.2 Mapped subroutines or functions

Our data/work distribution model extends the typing system with a mapping to specify the mapping of all variables of that type. Because subroutines and functions are also typed, they can also be mapped onto a processor array. A function can be mapped directly by specifying an owner for its type, or it can be aligned with one of its parameters.

For mapped functions, all parameters must be aligned with the owner of the function. Directly mapped functions are typically the I/O functions, which can only be executed on an I/O processor or I/O processor set. An aligned function typically occurs when the compiler can figure out how to align a function with one of its parameters by inspecting its body. Alternatively, the user may align the function to one of its parameters to simplify the task of the align-engine. Both function and subroutine mappings are illustrated in Example 48.
Example 48: Mapped functions

The mapping of a function specifies how data from the actual expressions are copied into the dummy variables of the functions. It also specifies the active processor set that is involved in the evaluation of the body. It does not specify how the executable code of the body of the function is mapped. Because the framework is based on an SPMD programming model, the executable code of the body is always distributed onto $P_{world}$.

For the expression containing the function call $gcd(expr_1, expr_2)$, the compiler knows that $gcd$ is aligned with the owner of the actual expression $expr_1$, so it can copy the value of $expr_2$ into a temporary variable aligned with $expr_1$ and execute $gcd$ only on the owner of $expr_1$ (i.e. the owner of $gcd$).

Without these specifications, the compiler has to select a mapping for a subroutine or function, depending on the context it is used. Usually they are mapped onto $P_{world}$; That means, in case of the $write$ function that all processors are involved in collecting and storing the actual expressions, while only the I/O processor will actually execute the write function. All other processors do need dummy $write$ functions.

7.3 Compound Statements

By introducing a compound statement into the language, denoted with a BEGIN and END keyword pair, it is easier to identify a piece of control flow to attach ON clauses to or to add variables local to the compound statement. When a source-to-source compiler analyses an HPF program and has to introduce temporary variables, it can give feedback to the user about the scope of the variables by introducing appropriate compound statements in its output.

A compound statement can also be annotated with an INDEPENDENT directive. It tells the compiler that the given statements sequence within the BEGIN END pair can be executed in an arbitrary order. For instance, for the Red-Black solver as given in Example 49, the compiler is free to swap the sequence. If it can also proof that both function calls can be executed on two mutually exclusive processor sets, the two can be executed in parallel.

43 HPF-2 already uses the BEGIN and END keywords for specifying a scope of an ON clause. It is however only an annotation and not a language construct.
7.4 Local variables inside compound statements

By allowing local variables inside compound statements, with types that may depend on variables that are defined on entry of the compound statement, more powerful analysis techniques can be used. For instance, variable declarations can be 'restricted' to their actual scope, thereby simplifying the dead-live analysis of variables\(^{44}\). The types of the local variables can be attributed with either an owner attribute or a private attribute.

Example 50: Allocatables compared with local variables inside a compound statement

Example 50 shows two ways of specifying a distributed array, which shape is only known at runtime. The ALLOCATE statement must be preceded with a REDISTRIBUTE or REALIGN directive to allocate mapped arrays. Although both specifications will result in the same allocation, the second specification is to be preferred from an analysis point of view. The second specification assures that the type of the array \(A\) will never change as long as array \(A\) lives. The first specification allows a redefinition of the type of array \(A\), because a REDISTRIBUTE is an executable statement. Even worse, the redefinition might even depend on the control flow of the program.

By allowing only the second specification, we expect that most of the original envisioned use of REDISTRIBUTE is covered, without loosing the opportunity for powerful analysis.

\(^{44}\) The NEW annotation also allows
7.5 Schedule extensions

To allow users to explicitly specify where schedules must be initialized (inspector part) and where schedules must be used (executor part) three extensions to the language are proposed.

The first extension is a primitive type Schedule is used to type the user defined schedule variables. As with all variable names in Fortran90, schedules are local entities of class (1); therefore, a schedule may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Names of schedules obey the same rules for host and use association as other names in the long list in Section 12.1.2.2.1 of the Fortran 90 standard.

The second extension is an intrinsic ‘function’\textsuperscript{45} schedule, which specifies all details needed to initialize the schedule. It specifies the iteration space, the owner from which the data originates and the owner, which receives the data.

The third extension specifies the iteration space of a forall with an on clause containing the schedule to be used. The schedule type extension, the schedule intrinsic function, and the schedule application are shown in Example 51.

```
SCHEDULE SCHED
    | INSPECTOR
SCHED = SCHEDULE (I = ...) FROM ... TO ...

    | EXECUTOR PHASE
FORALL (I = ... ON SCHED)
    ...
END FORALL
```

Example 51: Schedule extensions

The user is responsible for the correct use of the schedule. For instance, the user must verify that the communication pattern has not changed between the schedule’s initialization and usage. The compiler only checks the correct type of the expressions used at the places where schedules are to be expected. The compiler also assumes that the complete body of the forall can be evaluated at the owners as specified by the schedule.

7.6 Extended mappings

The framework is designed around the distributions that are allowed in HPF, but it can allow mappings that are much more general. As long as it allows, specifying a mapping per dimension of an owner that can be inverted. Then the above described multiple mask absorptions can still be applied.

\textsuperscript{45} Strictly speaking, this is not a function because it can not be compiled separately and stored into a library, because there is no way to express for instance a triangular iteration space as a parameter of the function.
7.6.1 The identical distribution

A trivial distribution is the identical distribution. It is equal to the block(I) distribution, with the extra constraint that the extent of the distributed template is equal to the extent of the corresponding dimension of the processor array. In addition, a shadow specifier is not allowed for identical distributions. Because of these restrictions, the compiler can derive that there is a one-to-one relation between processor indices and owner indices. A separate IDENTICAL directive is used for this distribution.

A special index specifier @ can be used in identical distributed dimensions. The value is equal to the index of the corresponding processor dimension of the running processor\textsuperscript{46}. Together with the @ index specifier expressive programs can be written as is shown in Example 52.

<table>
<thead>
<tr>
<th>RREAL TotalTime,LocalTime(NUMBER_OF_PROCESSORS())</th>
</tr>
</thead>
<tbody>
<tr>
<td>!HPF$ DISTRIBUTED LocalTime(IDENTICAL)</td>
</tr>
<tr>
<td>LocalTime(@) = time()</td>
</tr>
<tr>
<td>LocalTime(@) = LocalTime(@) - time()</td>
</tr>
<tr>
<td>TotalTime = SUM(LocalTime)</td>
</tr>
</tbody>
</table>

Example 52: Usage of the index specifier @.

The example above shows how on each processor execution time can be measured independently for each processor. The total time spent is then calculated as the sum of all local times. Notice that the HPF programmer does not need to know how many processors are currently running, nor does he need to know the value of @ that will be substituted by the compiler when generating SPMD code.

The identical distribution gives the HPF programmer complete control over the generated communication and the allocation of extra memory to simulate shadow areas as shown in Example 53.

\textsuperscript{46} Notice the similarity of the @ symbol in owner attributes (see Section 4.4.1).
Example 53: Motivating example for the identical distribution.

In this motivating example, a (BLOCK, BLOCK) distribution is simulated by adding two IDENTICAL distributions, mapping directly onto a processor index. Now the programmer has complete control over the local elements $A(\cdot, \cdot; p_1, p_2)$ for a given processor $P(p_1, p_2)$. Even copying into the simulated shadow areas can be explicitly specified as shown in gray in Figure 25.

Figure 25: Layout of array $A$. 
When generating SPMD code, the compiler can easily deduce that the assignment is a local assignment with owner $P(p_1, p_2)$ and leaves the $i$ and $j$ loops untouched. A simple mask will surround the two inner loops. The two communication statements will be recognized and straightforward packing and assignment loop nests will be generated.

The programming style is more closely to programming in FORTRAN 77 with MPI calls, but is much more readable. The programmer can still view the address space as one global address space. Introducing the identical distribution makes the programmer more aware of the capabilities of HPF and that it can be as expressive as directly programming in FORTRAN 77 with MPI calls.

### 7.6.2 HPF2 extended mappings

Most of the approved extensions of HPF 2.0 can be supported directly within our framework like a distribution onto a processor subset, the ON HOME clause, the SHADOW directive, and the generalized block distribution (specified with the GEN_BLOCK directive). The generalized block distribution allows a blocksize that depends on the processor index, i.e. $m_0(p)$ instead of $m_0$. Our model goes even further and allows a generalized block cyclic distribution gen_block_cycle.

The approved new distribution format indirect (specified with the INDIRECT directive) allows a one-dimensional array as a lookup table that associates an index of the distributed array with the index of the processor element. The extent of the map array must be equal to the extent of the dimension of the data array in which it occurs. In Example 54, the different distribution extensions are shown together with their derived owners.

```
!HPF$ PROCESSORS P(50)
!HPF$ TEMPLATE  T(100)
REAL A(50), B(100), M(50)
INTEGER MAP(100)

!HPF$ DISTRIBUT E A(BLOCK) ONTO P(1:7:2)  ! A: mapped (i) on P(block(i,m))
!HPF$ DISTRIBUT E B(GEN_BLOCK(M)) ONTO P ! B: mapped (i) on P(gen_block(i,m[0]))
!HPF$ DISTRIBUT E T(INDIRECT(MAP)) ONTO P ! T: mapped (i) on P(map(i))

!HPF$ ON HOME (T(I))  ! on P(map(i))
DO I = 1,100
    ...
END DO
```

**Example 54: Distribution extensions**

We claim however that an array $A(I:N)$ with an indirect distribution can be emulated by an array $A'(I:N)$ with a gen_block distribution, that is referenced with a proper permutation array perm(I:N) as shown in Example 55.
Example 55: Emulating an INDIRECT distribution.

The blocksize of processor \( p, m(p) \), is equal to the number of elements of \( \text{map}(i) \) that have a value equal to \( p \), i.e. \( m(p) = |\{ i | \text{map}(i) = p \}| \). The permutation array \( \text{perm} \) is a permutation that is needed to transform the unsorted array \( \text{map} \) into a sorted array \( \text{map}' = /0,0,0,1,1,1,2,2/ \). Notice that several permutation arrays can be found that will sort \( \text{map} \) as described above. The element \( \text{A}(i) \) then corresponds with \( \text{A}'(\text{perm}(i)) \) and both reside on the same processor as specified, but the local memory layout of \( \text{A} \) and \( \text{A}' \) might be completely different.

A requirement for letting the HPF programmer have complete control over how individual elements are stored, is that the HPF compiler stores consecutive elements of \( \text{A}' \) on a single processor in consecutive memory locations.
Figure 26: An irregular two-dimensional mesh.

As an example an irregular two-dimensional mesh mesh2D is given, partitioned over four processors (indicated by the four shades of gray) as shown in Figure 26. Mesh2D describes a two-dimensional grid that is used in a Computational Fluid Dynamics (CFD) model. It models water that is flowing from left to right and has to pass the corner placed in the center. Local grid refinement is applied to increase the accuracy of the model at places where needed.

Example 56 gives a code fragment where mesh2D is accessed in different contexts: a single access, or within a volume assignment. The example also uses an artificial expression $i \times j$ that needs the index $(i, j)$ of the iteration space.

```hpf
! non distributed version

  ... MESH2D(i,j) ...

  FORALL (i = ._; j = ._)
    MESH2D(i,j) = MESH2D(i,j) + MESH2D(i,j+1) + i * j
  END FORALL
```

Example 58: Example code using a two-dimensional mesh
To get a distributed version the code needs to be slightly rewritten, because there is no way to express a two-dimensional *indirect* mapping onto a one-dimensional processor array. Let a one-dimensional mesh $mesh1D$ be mapped as given in Example 57. Let $g2off(i, j)$ be the array that can be used as a global-to-offset function, i.e. $l = g2off(i, j)$, where $l$ is the local index of the one-dimensional $mesh1D$ and $(i, j)$ the index of the two-dimensional mesh $mesh2D$. Let $off2g$ be the array that can be used as a offset-to-global function, i.e. $(i, j) = off2g(l)$, where $l$ is the local index of the one-dimensional $mesh1D$ and $(i, j)$ the index of the two-dimensional mesh $mesh2D$. The array $M(p)$ evaluates to the number of nodes of $mesh1D$ owned by processor $p$.

```
INTEGER, DIMENSION(NR_OF_PROCESSORS()) :: M

!HPFS DISTRIBUTE MESH1D(GEN_BLOCK(M))
!HPFS ALIGN (1) WITH MESH1D(1) :: ind, off2g

- MESH1D(g2off(i, j)) =

  FORALL (i = ...; j = ...)
  ind(g2off(i, j)) = g2off(i, j + 1)
  END FORALL

  FORALL (l = ...)
  MESH1D(l) = MESH1D(l) + MESH1D(ind(l)) + off2g(l)*i + off2g(l)*j
  END FORALL
```

Example 57: Rewritten version of Example 56 using a one-dimensional mesh.

A single reference to a node of $mesh2D$ is equivalent to $mesh1D(g2off(i, j))$ in the rewritten example. Inside volume assignments like a forall, the original two-dimensional iteration space $(i, j)$ can be rewritten as a one-dimensional iteration space. The relation between two $mesh2D$ references can be expressed as a relation between two $mesh1D$ references. The original values of $i$ and $j$, corresponding to a point in the two-dimensional iteration space can be found with the offset-to-global array $off2g$.

Because the distribution is expressed as a *gen_block* distribution, the compiler will recognize the FORALL statement as a semi-regular assignment. It therefore can generate an efficient communication schedule, where it can absorb a single owner. If the same distribution was expressed as an *indirect* mapping, even the single owner can not be absorbed, thereby resulting in less efficient code.

Even if arbitrary mappings would have been supported in our compilation system as *indirect* distributions, it would have a major impact on the performance in those cases where mappings are inherited. In that case the compiler must for instance, assume the more general *indirect* distribution, instead of a *block_cyclic* distribution. Because an *indirect* distribution can also be expressed as a *gen_block* distribution, the compiler can rely on the *gen_block_cyclic* distribution as a default distribution.
7.7 Restrictions to the HPF 2.0 Language

A number of approved extensions have been proposed to the official HPF 2.0 standard. Moreover, several research groups have proposed even more extensions to be included in the standard. Most notably the HPF+ [15] and the JAHPF [35] efforts. However, of equal importance are the restrictions to the current language definition that can be defined. The restrictions must be chosen such that the expressiveness of the language is not reduced, the underlying model is more consistent, and the compiler can deduce stronger analysis results.

Because we have treated data-mappings as an extension to the data type of HPF arrays and scalars, all mapping extensions need to behave as type extensions. The compilation system must be sure that a type does not change during the lifetime of variables of that type.

For instance, REALIGN and REDISTRIBUTE directives are allowed on a variable, therefore its type can change during its lifetime. This would force the compiler to inspect the control flow to find out the reaching type definitions of a variable (if possible). It would be better to only use the REALIGN or REDISTRIBUTE directive for newly introduced variables. The type specification (shape as well as mapping) may then depend on values when entering the scope of the type definition (see Section 7.4). Redistributing or realigning of already mapped variables will not avoid extra copying, the main reason why realign and redistribute were invented.

Normally a type definition may be based on previously defined types and this will result in a hierarchy of types, normally a tree. In HPF, it is also allowed to build a hierarchy of mappings (a root mapping and a list of immediate alignments). However modifying the mapping of the root will result in a complete re-mapping of all variables, which are derived from that root mapping. It also results in side effects on the type of other variables.

That means special run-time administration is needed to trace all type dependencies. This contradicts with everything a type system is designed for, namely an internal compilation model to help the compiler reason about types of variables and expressions, the correct use of variables and functions, and type equivalence.

So in short, no dynamic mappings should be allowed as proposed in the approved extensions in the HPF 2.0 standard. As an alternative, a less restrictive static mapping must be allowed. The static mapping may depend on values outside the type's scope, but is frozen (static) inside the type's scope.

7.8 Related work

Because we have modeled the data-parallel mappings as type extensions, and thereby putting the extension in the heart of the compilation system, more restrictions on the usage of data mappings are needed as described in Section 7.7. Most of the other research groups handle data-parallel mappings in a run-time system, capable of supporting all kinds of mappings and re-mappings. Most notably are the Fortran D system [24] and the PARTI libraries [69]. A drawback of this
approach is that certain communication patterns need to be recognized by the compiler to call special library functions. These communication patterns are very restricted, and if none can be found, a very poorly performing fallback option needs to be selected.

This will result in HPF users trying to find out how their compiler thinks and why it selected that particular transformation. This contradicts with one of the design criteria of HPF that performance must be portable between different machines and therefore also between different compilers.
8 Experimental results

Evaluating the performance of HPF implementations in general is difficult. The HPF language is designed such that the HPF programmer may rely on a simple performance model: accessing local variables is much more efficient than accessing remote values. The HPF programmer can therefore influence the amount of local and remote accesses with proper alignments between variables. The structure of a program will prescribe the best data distributions to be used.

For an HPF programmer it is not acceptable that different compilers will give different performance results for the same program on the same machine. The performance itself may not be much larger than what the simple model would predict. It is also not acceptable that small changes in data distributions, and thereby small changes in the number of local and remote accesses, would result in big differences in performance.

From a compiler point-of-view, all kinds of programs need to be compiled as efficiently as possible. In particular it is not completely known which kind of alignments and distributions are likely to occur in practice. For the moment, we must therefore assume that all alignments and distributions are equally likely to occur, and we must be able to generate efficient code for each case.

The HPF compiler described in this thesis is currently not implemented. Its predecessor, the PRE-HPF compiler is implemented as part of the FUTURE project [26]. It supports only a small part of the model described in this thesis. It mainly implements an efficient lowering of a basic assignment. Extending the functionality of the PRE-HPF compiler, such that it complies with the design as described in this thesis would take several person-years to implement. It would require knowledge scattered over people of several companies, which are now assigned to other projects.

However, to show the potential of the described compiler model we have used the PRE-HPF compiler directly or indirectly by simulating the expected behavior of a real compiler through modifying the C code as generated by the PRE-HPF compiler. In the description of the experiments, we will clearly identify when we have used the modified PRE-HPF compiler. In the PRE-HPF compiler, most of the functionality of the alignment engine is not implemented. Mainly the Owner Computes Rule is applied, and where possible, stencil assignments are used. For two-owned assignments where the complete rhs is owned by a single processor the Sender Computes Rule is applied. The sending owner will evaluate the complete rhs before sending the results to the receiving owner.

For the lowering of basic assignments, multiple mask-absorption is implemented in the PRE-HPF compiler. Despite these shortcomings, several tests with the PRE-HPF compiler can be defined which shows the potential of our new data-parallel compilation system.

In order to get more insight in the performance of the compiler and the behavior of the techniques used, the PRE-HPF compiler was set against the commercial available PGI-HPF compiler and the freeware GMD-HPF compiler in a benchmark suite. The TNO benchmark suite is used to
EXPERIMENTAL RESULTS

examine the behavior of the HPF compilers in more detail than can be done with the publicly available benchmarks.

8.1 The Test Environment

8.1.1 Hardware: the Distributed ASCI Supercomputer

The hardware used in the test environment is the Distributed ASCI Supercomputer (DAS)[22]: a 200-node wide-area distributed system built out of four Myrinet-based Pentium Pro clusters. The Distributed ASCI Supercomputer is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI). The DAS machine is used for research on parallel and distributed computing by five Dutch universities: Vrije Universiteit Amsterdam, University of Amsterdam, Delft University of Technology, Leiden University, and the University of Utrecht.

DAS consists of four clusters, located at the first four universities. The first cluster (at the VU) contains 128 nodes, the other three clusters have 24 nodes (200 nodes in total). The operating system the DAS runs is RedHat Linux.

Each node contains:

- A 200 Mhz Pentium Pro (Siemens-Nixdorf D983 motherboard)
- 64 MB EDO-RAM in DIMM modules (128 MB for the clusters in Leiden and VU)
- A 2.5 GByte local disk
- A Myrinet interface card
- A Fast Ethernet interface card

The nodes within a local cluster are connected by a Myrinet SAN network, which is used as high-speed interconnect, mapped in user-space. In addition, Fast Ethernet is used as OS network (file transport). The four local clusters are connected by wide-area ATM links, so the entire system can be used as a 200-node wide-area distributed cluster. The topology is a 3D-mesh, in which dimension order routing is used.

The DAS topology

Each DAS cluster internally uses two networking technologies: 100 Mbit/sec FastEthernet for regular OS related traffic (NFS, telnet, rsh) and 1.2 Gbit/sec (full duplex) Myrinet SAN network for low-latency, high-bandwidth user-level communication.

47 http://www.cs.vu.nl/das
The four DAS clusters are interconnected over the SurfNet4 ATM network, which offers lower latency and higher bandwidth than the standard Internet connections between the sites. Every site has a special gateway host, which routes inter-cluster packets from the FastEthernet network over the wide area ATM links. It is important to mention that the links are actually Constant Bitrate (CBR) virtual circuits, offering guaranteed and reliable performance. This makes the behaviour and run time of applications run on the wide-area DAS much more predictable than would otherwise be the case.

They have a capacity of 6 Mbit/sec. The roundtrip latency between the sites is between 1.5 and 3.7 msec, mainly depending on the distance.

![Figure 27: A snapshot of the DAS cluster at the "Vrije Universiteit"](image)

A snapshot of the cluster at the Vrije Universiteit, clearly showing the Myrinet cables. For our tests, we will only use a small amount of the nodes from the Vrije Universiteit's cluster. There-
fore, the wide-area ATM part of the network is not used, but is only mentioned here for completeness.

8.1.2 Software: the HPF compilers

The compilers used for the benchmark tests are a commercial compiler, our PRE-HPF prototype compiler, and a freeware compiler. We selected the Portland Group Inc. HPF compiler as a commercial compiler because it is available on almost every parallel system. The ADAPTOR compiler is selected as the freeware compiler because it can easily be ported onto the DAS system.

The compiler-flags for each of these compilers are set to optimize the programs for speed and are given below:

The PGI-HPF Compiler: Portland Group Inc. HPF compiler, pghpf Release 3.1-2
- -Mmpi, MPI-library is used for communication.
- -03, optimization is required.
- -Mextend, long lines in the program code are allowed.
- -Mnofree Fortran-77 format is to be used.

The PRE-HPF Compiler: prehpf internal prototype
- -u, no fetch before store needs to be enforced.
- -x, Fortran-77 format is to be used.
- -O3 optimization is required.

The GMD-HPF Compiler: The ADAPTOR gmdhpf compiler (version 7.0 dec 1999).
- -O

All parallel compilers use the same Message Passing Interface (MPI) library as a communication layer. The MPI library is however specially optimized for the DAS communication network.

8.2 The TNO-Benchmarks

The goal of these benchmarks is to get detailed information on how the compilers perform on specific details of the SPMD code generation. The focus is on the efficient SPMD-lowering of Basic Assignments.
The benchmark test suite will test different compilers on small HPF programs (kernel programs). These kernel programs are generated from a program template. With a single template, kernel programs can be generated with a different assignment (A), iteration space (I), problem size (S), and mapped variables (D).

This gives an insight in the effectiveness of the various algorithms used by the compiler. A simple script file is used to generate test-files from the previous mentioned (A, I, S, D) four-dimensional test space, and thus obtaining a large number of test-files. These test-files are then executed and timed. The different template-files are:

- `init1D.tmpl`: initialization of a one-dimensional array.
- `copy1D.tmpl`: copying between two one-dimensional arrays.
- `init2D.tmpl`: non-rectangular initialization of a two-dimensional array.
- `shift1D.tmpl`: several shift assignments.
- `stencil1D.tmpl`: several stencil assignments.
- `SCR1D.tmpl`: assignment where sender computes rule applies.
- `semireg1D.tmpl`: several semi-regular assignments.

An example of a template file is given in Example 58. The lines prefixed with !S1!, !S2!, and !S3! are three lines that specify the size of the benchmark. The lines prefixed with !D1!, !D2!, and !D3! are three lines that specify the distribution of the used arrays. The lines prefixed with !A1! up to !A6! are the different assignments that are to be tested.
PROGRAM sample

!S1! INTEGER, parameter :: N=1000, imaxs=1
!S2! INTEGER, parameter :: N=500, imaxs=2
!S3! INTEGER, parameter :: N=250, imaxs=4
INTEGER, parameter :: imaxa=10

!HPFS$ PROCESSORS P(1,NUMBER_OF_PROCESSORS())

!D1!!HPFS$ DISTRIBUTED (BLOCK, BLOCK ) ONTO P :: A,B
!D2!!HPFS$ DISTRIBUTED (CYCLIC, CYCLIC ) ONTO P :: A,B
!D3!!HPFS$ DISTRIBUTED (CYCLIC(4),CYCLIC(4)) ONTO P :: A,B

FORALL (k=1:N,i=1:N) B(i,k) = real(i)/real(N)

tim1 = time_us()

DO iter = 1,imaxs*imaxa

CALL values(N,iter,1,u,s)

!T1! FORALL (k=1:u:s,i=1:u:s)
!T2! FORALL (k=1:u:s)

!T1! FORALL (i=k:u:s)
!T2! A(i,k) = 1
!T3! A(i,k) = B(i,k)
!T4! A(i,k) = B(i,k+1)
!T5! A(i,k) = B(i,k+10)
!T6! A(i,k+10) = B(i,k)

!T7! END FORALL

END FORALL

END DO

tim2 = time_us()

END PROGRAM

Example 58: A sample template (sample.tmpl).

To generate a test program from the template, the template name and a coordinate in the four-dimensional test space is needed. For instance, a test program derived from sample.tmpl for problem size S=2, distribution D=3, iteration space I=2, and assignment A=4 is denoted as sample-S2-D3-I2-A4.hpf and is shown in Example 59. The test program can be derived by removing the ‘!*!’ comment prefix form each line which contains one of the test space coordinates. Lines containing a ‘!*!’ comment prefix but none of the space coordinates can simply be removed.
This provides a simple but powerful benchmarking environment to obtain a complete test space.

To avoid effects of other optimizations, like Loop Interchange or Invariant Code Motion (ICM), the benchmarks are written such that those optimizations can not be applied. It avoids that the results of the various compilers are incomparable.

To avoid depending on loop interchange, the FORALL as used in Example 58 is already in such an order that the compiler can simply generate a loopnest by traversing the iterators from left to right. The leftmost iterator \( k \) will be the iterator of the outer loop and the rightmost iterator \( i \) the iterator of the inner loop. This guarantees that the fastest running index \( i \) references consecutive elements of the array \( a \) in local memory.

The additional call to the subroutine \texttt{values} assures that the compiler can not hoist out (possibly) expensive loop bounds reduction and initialization code. Given these two additional provisions a fair comparison on the efficiency of basic assignments can be made between the two parallel compilers.

### 8.2.1 Initializing an array

To get any idea about the overhead that is introduced by mask absorption we will use a simple one-dimensional array assignment that does not need communication. We are mainly interested in the performance when 1) different alignments, 2) different kinds of templates, and 3) different enumerations (row-wise versus column-wise) are used.

We distinguish four kinds of templates:
• A block distributed template, when there is only one row in the template.

• A block-like distributed template, when there is in the template only a small amount of rows compared to the number of columns.

• A cyclic-like distributed template, when there is in the template only a small amount of columns compared to the number of rows.

• A cyclic distributed template, when there is only one column in the template.

The test template initIDtmpl is used to measure the overhead of generating the proper local element sets of a simple assignment A(i) = 1, for 4 different D1:D4 distributions. D1 is a block distribution. D2 is a 'block-like' distribution where the blocksize is chosen in such a way that the number of rows allocated locally is always four, independent on the number of processors. D3 is a 'cyclic-like' distribution where the number of columns is always four. D4 is a cyclic distribution.

```fortran
PROGRAM initID
    INTEGER, parameter :: N=100000,M=N/(4*NUMBER_OF_PROCESSORS()),imaxax=1000,
    INTEGER :: i,iter,l,u,s
    REAL :: A(N),tim1,tim2,time_us
!
HPFS PROCESSORS P(NUMBER_OF_PROCESSORS())
!
| D 1, 2, 3, 4 | HPFS TEMPLATE T(N)
| D 5, 6, 7, 8 | HPFS TEMPLATE T(N+2)
| D 9,10,11,12 | HPFS TEMPLATE T(3*N+7)
| D 1, 5, 9 | HPFS DISTRIBUT T(BLOCK ) ONTO P
| D 2, 6, 10 | HPFS DISTRIBUT T(CYCLIC(N)) ONTO P ! block-like
| D 3, 7, 11 | HPFS DISTRIBUT T(CYCLIC(4)) ONTO P ! cyclic-like
| D 4, 8, 12 | HPFS DISTRIBUT T(CYCLIC ) ONTO P
!
| D 1, 2, 3, 4 | HPFS ALIGN (i) WITH T( i ) :: A
| D 5, 6, 7, 8 | HPFS ALIGN (i) WITH T( i+2 ) :: A
| D 9,10,11,12 | HPFS ALIGN (i) WITH T(3*i+7) :: A
!
tim1 = time_us()
    DO iter = 1, imaxax
        CALL values(N,iter,l,u,s)
        FORALL (i=1:u:s)
            A(i) = 1.0
        END FORALL
    END DO
    tim2 = time_us()
    print tim2-tim1
END PROGRAM 
```

Example 60: Init1Dtmpl

An additional DO loop is placed around the FORALL to keep the total measured time far away from the system clock resolution of 1ms.

122
To test the different enumeration schemes (row-wise or column-wise) the PRE-HPF compiler could not be used as is, because the PRE-HPF compiler was designed to use only column-wise enumeration. We used the PRE-HPF compiler to generate C code as output. The C code was then modified such that both enumeration schemes could be tested. The modified C code was compiled with the PGI C compiler (Release 3.1-2), to get the best possible comparable results.

The test is run for all three compilers using the init1D.tmpl template for $D = 1:4$ (denoted as init1D-D:1:4) as given in Example 60. The measured results in seconds are given in Table 2.

<table>
<thead>
<tr>
<th>Trend</th>
<th>Align A(i) with T(i)</th>
<th>Align A(i) with T(i+2)</th>
<th>Align A(i) with T(3:7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>bc-col bc_row/cy</td>
<td>pgc gmd</td>
<td>bc-col</td>
</tr>
<tr>
<td>block</td>
<td>1 184.29 5.75</td>
<td>5.75</td>
<td>5.97</td>
</tr>
<tr>
<td>2 91.98 2.26</td>
<td>2.28</td>
<td>2.70</td>
<td>2.17</td>
</tr>
<tr>
<td>3 61.18 1.26</td>
<td>1.30</td>
<td>1.54</td>
<td>1.01</td>
</tr>
<tr>
<td>4 45.01 0.76</td>
<td>0.76</td>
<td>1.27</td>
<td>0.76</td>
</tr>
<tr>
<td>5 36.64 0.61</td>
<td>0.61</td>
<td>0.93</td>
<td>0.61</td>
</tr>
<tr>
<td>6 30.94 0.51</td>
<td>0.51</td>
<td>0.76</td>
<td>0.51</td>
</tr>
<tr>
<td>block-like</td>
<td>1 56.24 22.66</td>
<td>6.16</td>
<td>56.24</td>
</tr>
<tr>
<td>2 27.99 6.40</td>
<td>2.51</td>
<td>28.00</td>
<td>6.96</td>
</tr>
<tr>
<td>3 18.61 2.38</td>
<td>1.90</td>
<td>18.64</td>
<td>3.20</td>
</tr>
<tr>
<td>4 13.94 1.14</td>
<td>1.43</td>
<td>13.94</td>
<td>1.15</td>
</tr>
<tr>
<td>5 11.15 0.92</td>
<td>1.15</td>
<td>11.15</td>
<td>0.92</td>
</tr>
<tr>
<td>6 9.29 0.77</td>
<td>0.95</td>
<td>9.29</td>
<td>0.77</td>
</tr>
<tr>
<td>cyclic-like</td>
<td>1 6.04 57.57</td>
<td>30.99</td>
<td>191.11</td>
</tr>
<tr>
<td>2 2.43 28.40</td>
<td>15.23</td>
<td>285.55</td>
<td>2.48</td>
</tr>
<tr>
<td>3 1.28 18.75</td>
<td>10.13</td>
<td>250.50</td>
<td>1.14</td>
</tr>
<tr>
<td>4 0.77 13.90</td>
<td>7.56</td>
<td>231.08</td>
<td>0.77</td>
</tr>
<tr>
<td>5 0.61 11.19</td>
<td>6.06</td>
<td>220.07</td>
<td>0.61</td>
</tr>
<tr>
<td>6 0.62 9.31</td>
<td>5.04</td>
<td>212.27</td>
<td>0.51</td>
</tr>
<tr>
<td>cyclic</td>
<td>1 6.27 185.09</td>
<td>6.27</td>
<td>8.68</td>
</tr>
<tr>
<td>2 2.47 91.35</td>
<td>2.46</td>
<td>4.33</td>
<td>120.30</td>
</tr>
<tr>
<td>3 1.27 60.78</td>
<td>1.30</td>
<td>2.89</td>
<td>79.95</td>
</tr>
<tr>
<td>4 0.75 45.52</td>
<td>0.80</td>
<td>21.59</td>
<td>89.85</td>
</tr>
<tr>
<td>5 0.61 36.41</td>
<td>0.63</td>
<td>1.73</td>
<td>47.97</td>
</tr>
<tr>
<td>6 0.51 30.35</td>
<td>0.53</td>
<td>1.44</td>
<td>39.88</td>
</tr>
</tbody>
</table>

Table 2: init1D: Initializing a one-dimensional array

The column labeled with np shows the number of processors running the test. The column labeled Tkind gives the kind of template the array was aligned with. The alignment itself is given in the second row. For each of these alignments, three columns are given for the HPF compiler as designed in this thesis, labeled (bc_col, bc_row, bl/cy). The column labeled bc_col gives the timings of a double loop column-wise enumeration (the result of a block cyclic mask absorption). The column labeled bc_row gives the timings of a double loop row-wise enumeration. The third column labeled with bl/cy gives the timings of a single loop block enumeration when the template is block distributed, respectively the timings of a single loop cyclic enumeration when the template is cyclic.

48 We assume that the low-level optimizations of the PGI HPF compiler and the PGI C compiler are better comparable than just using an arbitrary C compiler.
plate is cyclic distributed. The columns labeled with pgi and gmd are the timings of the PGI-HPF and GMD-HPF generated codes, respectively. The blank table entries denote that the program did either not compile or crashed at run-time. The gray table entries denote the best timings within a given template kind and alignment.

We will first concentrate on the timings of the PRE-HPF compiler. When looking at Table 2, many columns are the same. For instance, all timings of the direct alignment ‘Align A(i) with T(i)’ are almost the same to the timings of the shifted alignment ‘Align A(i) with T(i+2)’, as expected. The strided alignment ‘Align A(i) with T(3i+7)’ however gives totally different timings for the columns of the block-like, cyclic-like, and cyclic distributions. Because the stride is three the compiler might check, worst case three times as much rows or columns when enumerating bc_col or bc_row.\(^{49}\)

Whether the compiler knows that the distribution is block or cyclic or not does not matter for the performance, as long as the compiler enumerates bc_row for a block distribution and bc_col for a cyclic distribution. In fact, this is no surprise because the generated outer loop will only loop once and therefore will hardly generate extra loop overhead. On the other hand, when the compiler selects the wrong enumeration method, the inner loop would only loop once the worst-case loop overhead will be measured, as shown in the columns marked with the ellipses.

The generated code for block_cyclic enumeration allows a selection for a bc_col or bc_row enumeration at run-time. The run-time system can then always select the largest loop as the inner loop, i.e. bc_row for block-like template kinds and bc_col for cyclic-like template kinds. If it does so, the timings will be mostly the best, i.e. marked gray. The timings are independent of the alignment, and independent of the distribution, except for the block-like distribution.

For the block-like distribution, the best timings are shared between the PGI-HPF compiler and the PRE-HPF compiler. For a few processors, the PGI-HPF compiler performs best, because the local elements are stored row-wise and therefore the efficient stride-one memory access occurs. The PRE-HPF compiler timings correspond to the less efficient strided memory access, because it has a fixed column-wise storage of local elements\(^{50}\). If we would have selected a row-wise storage for the local elements the timings of the (bc_row, block-like) columns can be swapped with the timings of the (bc_col, cyclic-like) columns. The PRE-HPF compiler would then have out-performed the PGI-HPF compiler for all three alignments.

Looking at the PGI-HPF timings, we can conclude that it uses a fixed row-wise enumeration scheme and the local elements are stored row-wise. This results in the same timings for block templates as for the PRE-HPF compiler and better timings for the block-like templates, due to the row-wise local storage. If the PRE-HPF compiler was designed with row-wise storage the

\(^{49}\) The number of rows in a column-wise enumeration that needs to be checked is \(a/g\), where \(a\) is the alignment stride and \(g = \gcd(a, m)\), which can vary between \(l\) and \(a\) itself.

\(^{50}\) In contrary to the enumeration, the allocation method (row-wise or column-wise) can not change during the lifetime of an array.
above mentioned columns can be swapped and would have again outperformed the PGI-HPF compiler by a factor between 1.5 and 4.

The PGI-HPF compiler produces bad timings for template types where it should have switched over to column-wise enumeration, like in the cyclic-like templates. The PGI-HPF timings of a cyclic template are also strongly dependent on the alignment used. It varies from only twice as bad for a direct alignment up to 40 times as bad for a strided alignment. The fact that the PGI-HPF compiler is not performing as good as our compiler is less severe as the fact that it depends so strongly on the alignment used.

Looking at the GMD-HPF timings, we see that in many cases the program did not compile or produced a run-time error. For the block template kinds, where we did find timings, the GMD-HPF compiler is as efficient or sometimes (in case of 2 or 3 processors) slightly better than the PRE-HPF compiler. The other GMD-HPF timings, i.e. the cyclic-like and the cyclic template kinds of the first alignment, are a factor 30 to 400 larger than the best timings.

We may conclude that we have found an enumeration method that is independent on the alignment and performs equally well for each distribution.

### 8.2.2 Non-Rectangular iteration spaces

In this benchmark, we want to measure the loop overhead when non-rectangular iteration spaces are used. To avoid performance differences due to communication we will concentrate on the initialization of a two dimensional array. Three kinds of non-rectangular iteration spaces will be tested: 1) iterating over a diagonal of a two-dimensional array, 2) iterating over the upper triangle of a two-dimensional array, and 3) iterating over a convex shaped iteration space.

The same distributions (block, block-like, cyclic-like, cyclic) will be used for both dimensions of the mapped array. The test template is shown in Example 61.
Iterating over a diagonal

Strictly speaking, a diagonal iteration space is rectangular, only multiple masks are absorbed to determine the correct loop bounds. The timings of the iterations over a diagonal are given in Table 3. The distribution mentioned in the columns is the distribution of the second dimension, those mentioned in the rows from the first dimension of the mapped array $A$.

The timings of all compilers are placed side by side. The table entries that show the best performance figures are marked gray. When table entries have comparable timings, both are marked gray.
In case of a \( (\text{block, block}) \), \( (\text{block-like, block-like}) \) and a \( (\text{block-like, cyclic-like}) \) template distribution, both compilers perform equally well. In one case, the \( (\text{block-like, block}) \) distribution, the PGI-HPF compiler clearly outperforms the PRE-HPF compiler, because the PGI-HPF compiler uses a row-wise storage and has therefore an advantage\(^{51}\). In all other cases, the PRE-HPF compiler outperforms the PGI-HPF compiler, ranging from a factor of 1.2 \( (\text{block-like, cyclic}) \) up to 9 times \( (\text{block, cyclic-like}) \).

<table>
<thead>
<tr>
<th>diag np</th>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pg i</td>
<td>pre</td>
<td>gmd</td>
<td>pg i</td>
</tr>
<tr>
<td>block 1</td>
<td>10.45</td>
<td>11.93</td>
<td>27.66</td>
<td>117.71</td>
</tr>
<tr>
<td>2</td>
<td>5.11</td>
<td>5.76</td>
<td>13.34</td>
<td>57.85</td>
</tr>
<tr>
<td>3</td>
<td>3.24</td>
<td>3.76</td>
<td>8.91</td>
<td>38.62</td>
</tr>
<tr>
<td>4</td>
<td>2.18</td>
<td>2.63</td>
<td>6.59</td>
<td>28.98</td>
</tr>
<tr>
<td>5</td>
<td>1.88</td>
<td>1.90</td>
<td>5.23</td>
<td>23.08</td>
</tr>
<tr>
<td>6</td>
<td>1.44</td>
<td>1.54</td>
<td>4.28</td>
<td>19.16</td>
</tr>
<tr>
<td>block-like 1</td>
<td>11.37</td>
<td>44.75</td>
<td>115.29</td>
<td>117.98</td>
</tr>
<tr>
<td>2</td>
<td>5.62</td>
<td>22.41</td>
<td>57.76</td>
<td>52.06</td>
</tr>
<tr>
<td>4</td>
<td>2.80</td>
<td>11.31</td>
<td>29.15</td>
<td>29.68</td>
</tr>
<tr>
<td>5</td>
<td>2.26</td>
<td>8.74</td>
<td>23.50</td>
<td>23.68</td>
</tr>
<tr>
<td>cyclic-like 1</td>
<td>36.72</td>
<td>13.58</td>
<td>211.38</td>
<td>142.59</td>
</tr>
<tr>
<td>2</td>
<td>18.28</td>
<td>6.63</td>
<td>272.07</td>
<td>70.99</td>
</tr>
<tr>
<td>3</td>
<td>12.07</td>
<td>4.46</td>
<td>244.73</td>
<td>47.24</td>
</tr>
<tr>
<td>4</td>
<td>8.87</td>
<td>3.20</td>
<td>225.34</td>
<td>35.36</td>
</tr>
<tr>
<td>5</td>
<td>7.14</td>
<td>2.57</td>
<td>215.19</td>
<td>28.32</td>
</tr>
<tr>
<td>6</td>
<td>5.82</td>
<td>2.21</td>
<td>208.41</td>
<td>23.53</td>
</tr>
<tr>
<td>cyclic 1</td>
<td>12.40</td>
<td>11.52</td>
<td>43.87</td>
<td>116.55</td>
</tr>
<tr>
<td>2</td>
<td>6.28</td>
<td>5.91</td>
<td>100.55</td>
<td>58.38</td>
</tr>
<tr>
<td>3</td>
<td>4.18</td>
<td>3.86</td>
<td>66.83</td>
<td>39.00</td>
</tr>
<tr>
<td>4</td>
<td>3.15</td>
<td>2.86</td>
<td>50.09</td>
<td>29.27</td>
</tr>
<tr>
<td>5</td>
<td>2.49</td>
<td>2.31</td>
<td>39.98</td>
<td>23.46</td>
</tr>
<tr>
<td>6</td>
<td>2.10</td>
<td>1.95</td>
<td>33.30</td>
<td>19.61</td>
</tr>
</tbody>
</table>

Table 3: Ini2D-A1: Diagonal iteration space.

The timings of the code generated by the GMD-HPF compiler does not even come close to the timings of the other two compilers.

The HPF compiler described in this thesis would, once implemented, produce better timings than the PRE-HPF compiler because it will generate more efficient global-to-local functions. More importantly, the run-time system will be able to select the proper enumeration order (row-wise or column-wise) on the fly. We expect that our HPF compiler will then outperform the PGI-HPF compiler in all cases.

\(^{51}\) Notice that the PRE-HPF compiler has a fixed column-wise enumeration.
Triangular iteration space

The timings for the triangular iteration space are given in Table 4. The differences in performance now become even clearer. In all cases, the PGI-HPF compiler is outperformed by the PRE-HPF compiler with a factor ranging from 1.6 (block, block) up to 40 (cyclic-like, cyclic-like). The main reason why the PGI-HPF compiler is performing so badly is the fact that it falls back onto a less efficient element-wise communication scheme. All communication takes place in the inner loop and each element is communicated individually. This will result in many small messages each having its own communication startup time. As shown in the table below this will kill the possible performance gain that could be achieved.

<table>
<thead>
<tr>
<th>triang</th>
<th>np</th>
<th>block</th>
<th></th>
<th>block-like</th>
<th></th>
<th>cyclic-like</th>
<th></th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>pre</td>
<td>gmd</td>
<td>pre</td>
<td>gmd</td>
<td>pre</td>
<td>gmd</td>
<td>pre</td>
</tr>
<tr>
<td>block</td>
<td>1</td>
<td>30.76</td>
<td>8.06</td>
<td>30.68</td>
<td>116.56</td>
<td>46.14</td>
<td>116.85</td>
<td>8.22</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.15</td>
<td>5.69</td>
<td>7.19</td>
<td>92.86</td>
<td>14.86</td>
<td>92.88</td>
<td>5.94</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6.39</td>
<td>4.38</td>
<td>5.11</td>
<td>76.19</td>
<td>7.97</td>
<td>76.21</td>
<td>4.38</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.23</td>
<td>3.44</td>
<td>3.82</td>
<td>66.18</td>
<td>5.27</td>
<td>66.18</td>
<td>3.45</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.84</td>
<td>2.83</td>
<td>3.23</td>
<td>59.68</td>
<td>3.91</td>
<td>59.67</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4.43</td>
<td>2.34</td>
<td>2.73</td>
<td>55.11</td>
<td>3.06</td>
<td>55.11</td>
<td>2.36</td>
</tr>
<tr>
<td>block-like</td>
<td>1</td>
<td>112.35</td>
<td>13.40</td>
<td>156.19</td>
<td>48.32</td>
<td>158.94</td>
<td>15.68</td>
<td>141.86</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>98.17</td>
<td>6.64</td>
<td>122.25</td>
<td>15.20</td>
<td>122.03</td>
<td>7.07</td>
<td>113.13</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>86.26</td>
<td>4.44</td>
<td>104.52</td>
<td>8.30</td>
<td>103.68</td>
<td>4.75</td>
<td>97.40</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>79.98</td>
<td>3.07</td>
<td>95.10</td>
<td>5.02</td>
<td>93.97</td>
<td>3.29</td>
<td>89.13</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>76.11</td>
<td>2.64</td>
<td>89.37</td>
<td>4.11</td>
<td>88.03</td>
<td>2.84</td>
<td>84.08</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>73.49</td>
<td>2.17</td>
<td>85.44</td>
<td>3.22</td>
<td>83.97</td>
<td>2.36</td>
<td>80.65</td>
</tr>
<tr>
<td>cyclic-like</td>
<td>1</td>
<td>115.45</td>
<td>12.27</td>
<td>17.82</td>
<td>158.83</td>
<td>48.55</td>
<td>156.48</td>
<td>24.35</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>96.06</td>
<td>4.90</td>
<td>51.35</td>
<td>115.66</td>
<td>13.32</td>
<td>115.18</td>
<td>5.18</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>84.53</td>
<td>3.50</td>
<td>34.31</td>
<td>98.10</td>
<td>7.23</td>
<td>97.91</td>
<td>3.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>78.80</td>
<td>2.35</td>
<td>25.95</td>
<td>89.35</td>
<td>4.38</td>
<td>89.68</td>
<td>2.46</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>75.43</td>
<td>1.96</td>
<td>20.82</td>
<td>84.13</td>
<td>3.36</td>
<td>84.50</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>73.20</td>
<td>1.61</td>
<td>17.43</td>
<td>80.64</td>
<td>2.64</td>
<td>80.73</td>
<td>1.72</td>
</tr>
<tr>
<td>cyclic</td>
<td>1</td>
<td>88.37</td>
<td>21.04</td>
<td>21.71</td>
<td>137.58</td>
<td>48.19</td>
<td>140.08</td>
<td>23.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>80.48</td>
<td>4.94</td>
<td>51.04</td>
<td>104.74</td>
<td>13.80</td>
<td>106.19</td>
<td>5.04</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>68.79</td>
<td>3.25</td>
<td>33.97</td>
<td>85.07</td>
<td>7.14</td>
<td>86.99</td>
<td>3.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>62.91</td>
<td>2.26</td>
<td>25.59</td>
<td>75.27</td>
<td>4.29</td>
<td>77.43</td>
<td>2.31</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>59.51</td>
<td>1.92</td>
<td>20.44</td>
<td>69.42</td>
<td>3.30</td>
<td>71.73</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>57.30</td>
<td>1.50</td>
<td>17.05</td>
<td>65.51</td>
<td>2.52</td>
<td>67.94</td>
<td>1.61</td>
</tr>
</tbody>
</table>

Table 4: Init2D-A2: Triangular iteration space.

The GMD-HPF compiler is capable of aggregating messages over the inner loop, instead of aggregating over both loops. The GMD-HPF compiler therefore performs better than the PGI-HPF compiler.
Convex shaped iteration space

With a convex shaped iteration space, we mean all shapes that can be expressed as a double loop where the inner loop bounds depend in an arbitrary way on the outer loop’s iterator. The timings of this test can be found in Table 5. Again, the table entries with the best (or comparable) performance figures are marked gray.

<table>
<thead>
<tr>
<th>n(l):u(i) np</th>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pg i</td>
<td>pre</td>
<td>gmd</td>
<td>pg i</td>
</tr>
<tr>
<td>block</td>
<td>1</td>
<td>45.48</td>
<td>15.49</td>
<td>47.77</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12.22</td>
<td>7.64</td>
<td>9.79</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9.34</td>
<td>5.10</td>
<td>6.23</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.81</td>
<td>3.82</td>
<td>4.42</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7.28</td>
<td>3.06</td>
<td>3.64</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6.80</td>
<td>2.50</td>
<td>3.05</td>
</tr>
<tr>
<td>block-like</td>
<td>1</td>
<td>230.58</td>
<td>44.36</td>
<td>306.13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>187.70</td>
<td>9.65</td>
<td>231.68</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>164.84</td>
<td>6.27</td>
<td>197.58</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>153.51</td>
<td>4.38</td>
<td>180.63</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>146.68</td>
<td>3.64</td>
<td>170.52</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>142.17</td>
<td>3.03</td>
<td>163.78</td>
</tr>
<tr>
<td>cyclic-like</td>
<td>1</td>
<td>239.11</td>
<td>47.72</td>
<td>48.70</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>191.72</td>
<td>9.36</td>
<td>101.95</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>168.56</td>
<td>6.04</td>
<td>67.86</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>157.12</td>
<td>4.29</td>
<td>51.06</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>150.35</td>
<td>3.55</td>
<td>40.83</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>145.87</td>
<td>2.95</td>
<td>34.07</td>
</tr>
<tr>
<td>cyclic</td>
<td>1</td>
<td>179.34</td>
<td>48.93</td>
<td>50.11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>160.38</td>
<td>9.70</td>
<td>101.77</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>137.08</td>
<td>6.31</td>
<td>67.70</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>125.58</td>
<td>4.36</td>
<td>50.95</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>118.76</td>
<td>3.69</td>
<td>40.66</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>114.25</td>
<td>3.07</td>
<td>33.90</td>
</tr>
</tbody>
</table>

Table 5: Init2D-A3: Convex iteration space

Roughly, the same conclusions can be drawn as for the triangular case. In all cases, the PGI-HPF compiler is outperformed by the PRE-HPF compiler. Most of the timings have doubled because the iteration space has doubled. The overall performance of the GMD-HPF compiler lies between the PRE-HPF compiler and the PGI-HPF compiler.

8.2.3 Copying an array

To measure the communication overhead we have defined a simple test. Array $B$ is assigned to array $A$ for different distributions of $A$ and $B$ as shown in Example 62.
PROGRAM copy1D
INTEGER, parameter :: N=120000, imaxa=100, M=N/(4*NUMBER_OF_PROCESSORS())
INTEGER :: i, iter, l, u, s
REAL :: A(N), B(N), tim1, tim2, time_us
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!

Example 62: Copy1D.tmpl

For the exact figures see Table 6. The distribution of A is shown vertically, that of B is shown horizontally. The best timings for each distribution combination are marked gray.
<table>
<thead>
<tr>
<th>copyID</th>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>np</td>
<td>gpi</td>
<td>pre</td>
<td>gmd</td>
</tr>
<tr>
<td>block</td>
<td>1</td>
<td>7.35</td>
<td>6.10</td>
<td>6.01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.05</td>
<td>3.54</td>
<td>2.84</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.98</td>
<td>2.27</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.38</td>
<td>1.70</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.10</td>
<td>1.35</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.97</td>
<td>1.13</td>
<td>0.62</td>
</tr>
<tr>
<td>block-like</td>
<td>1</td>
<td>6.48</td>
<td>41.78</td>
<td>6.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7.18</td>
<td>55.12</td>
<td>3.22</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5.08</td>
<td>53.30</td>
<td>2.14</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.19</td>
<td>53.65</td>
<td>1.62</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.48</td>
<td>54.09</td>
<td>1.30</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4.73</td>
<td>53.55</td>
<td>1.07</td>
</tr>
<tr>
<td>cyclic-like</td>
<td>1</td>
<td>6.54</td>
<td>25.27</td>
<td>6.53</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>134.51</td>
<td>25.86</td>
<td>251.12</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>99.01</td>
<td>15.43</td>
<td>218.09</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>68.48</td>
<td>9.82</td>
<td>197.90</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>60.73</td>
<td>6.11</td>
<td>185.76</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>48.85</td>
<td>5.76</td>
<td>179.16</td>
</tr>
<tr>
<td>cyclic</td>
<td>1</td>
<td>6.12</td>
<td>7.46</td>
<td>6.41</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>463.99</td>
<td>15.43</td>
<td>250.38</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>349.04</td>
<td>12.66</td>
<td>216.36</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>240.71</td>
<td>8.46</td>
<td>196.59</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>217.77</td>
<td>8.68</td>
<td>186.45</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>175.50</td>
<td>6.95</td>
<td>184.45</td>
</tr>
</tbody>
</table>

Table 6: Copy1D: Copying a 1-D array

As expected, the PRE-HPF compiler is outperformed by the PGI-HPF compiler for the (block, block), (block, block-like), (block-like, block), and (block-like, block-like) combinations, because the PRE-HPF compiler is designed to use column-wise enumeration (see Section 8.2.1). The GMD-HPF compiler however, even outperforms the PGI-HPF compiler in the (block, block) case.

Likewise, as expected, the PGI-HPF compiler is outperformed by the PRE-HPF compiler for the (cyclic, cyclic), (cyclic, cyclic-like), (cyclic-like, cyclic), and (cyclic-like, cyclic-like) combinations, because the PGI-HPF compiler is designed to use row-wise enumeration (see Section 8.2.1). For seven out of the eight other cases, the PRE-HPF compiler outperformed the PGI-HPF and the GMD-HPF compiler. For five out of the seven cases where the GMD-HPF compiler did produce timing results, it outperformed the PGI-HPF compiler.

The PRE-HPF compiler uses a fixed and for some communication patterns even the worst possible schedule for communicating the individual messages. That is why some combinations of template distributions, for example the (block-like, cyclic-like) or the (block-like, block), do not scale at all or not very well.
At the time of implementing the PRE-HPF compiler, priority was given to send messages as soon as possible, not to the order in which messages were sent. As soon as a message was packed on the sending processor, it was sent to the receiving processor. Consequently, all processors will send their first message to the same processor, i.e. a network contention can occur. Furthermore, all receiving processors expect to receive messages in a certain order, resulting in unnecessary extra buffer copying in the run-time system.

The HPF compiler as described in this thesis, however, is designed to generate code that first packs all messages in one send buffer. Size information is communicated among the receiving processors, to enable them to allocate enough buffer space, and to receive a known number of messages in a random order. We expect to see the timings scale considerably better once this scheme is implemented.

### 8.2.4 Shifting an array

Shifting an array results into communication between two neighboring processors, a little communication for a block distribution, a lot for a cyclic distribution, and something in between for block cyclic distributions. The template used for this test is given in Example 63.

The PRE-HPF compiler does exploit the fact that only two processors are involved in receiving remote data, but does not use the stencil optimization as described in Section 6.1.4. Because the tested assignment is a two-owned assignment and no data needs to be replicated once received at the receiving owner, the compiler can generate code that does the local-to-local copy directly. This avoids packing and unpacking of data that is already locally available.

```fortran
PROGRAM shift1D
    INTEGER, parameter :: N=60000, imaxa=100, M=N/(4*NUMBER_OF_PROCESSORS())
    INTEGER :: i, iter, l, u, s
    REAL :: A(N), B(N), tim1, tim2, time_us
    !HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
    !D1!!HPF$ DISTRIBUTED (BLOCK     ) ONTO P :: A, B
    !D2!!HPF$ DISTRIBUTED (CYCLIC    ) ONTO P :: A, B
    !D3!!HPF$ DISTRIBUTED (CYCLIC(M)) ONTO P :: A, B
    !D4!!HPF$ DISTRIBUTED (CYCLIC(4)) ONTO P :: A, B
    tim1 = time_us()
    DO iter = 1, imaxa
        CALL values(N, iter, l, u, s)
        FORALL (i=l:u:s)
            !A1!
            A(i) = B(i+1)
            !A2!
            A(i) = B(i+10)
        END FORALL
    END DO
    tim2 = time_us()
    print tim2-tim1
END PROGRAM
```

Example 63: Shift1Dtmpl
The timings for the first assignment are given in Table 7. As expected the PRE-HPF compiler performs badly for the block-like distribution because it is designed to use column-wise enumeration, likewise for the PGI-HPF compiler for the cyclic-like distribution. For the other cases, the PRE-HPF compiler performs best, except for the performance of the parallel program running on one processor. The reason for that can be found in the elaborate code generation for the global-to-local function and the determination of the local stride.

<table>
<thead>
<tr>
<th></th>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>pgi</td>
<td>pre</td>
<td>gmd</td>
<td>pgi</td>
</tr>
<tr>
<td>1</td>
<td>0.57</td>
<td>0.62</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>2</td>
<td>0.27</td>
<td>0.23</td>
<td>0.28</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.30</td>
<td>0.14</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>4</td>
<td>0.13</td>
<td>0.09</td>
<td>0.13</td>
<td>0.09</td>
</tr>
<tr>
<td>5</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>0.08</td>
<td>0.07</td>
<td>0.09</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 7: Shift1D-A1: a(i) = B(i+1)

The GMD-HPF compiler can match up with the other two compilers in case of a block distribution. For the cyclic distribution, the GMD-HPF compiler performs better than the PGI-HPF compiler but worse than the PRE-HPF compiler.

When the stencil offset is increased to ten elements, we get the performance figures as shown in Table 8. Surprisingly the PGI-HPF compiler is now performing slightly better for the block distribution than it does when only one element was shifted, despite the fact that more elements need to be communicated among the processors. The performance of the PGI-HPF compiler for the block distribution now equals the performance of the PRE-HPF compiler.

<table>
<thead>
<tr>
<th></th>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>pgi</td>
<td>pre</td>
<td>gmd</td>
<td>pgi</td>
</tr>
<tr>
<td>1</td>
<td>0.52</td>
<td>0.60</td>
<td>1.10</td>
<td>0.53</td>
</tr>
<tr>
<td>2</td>
<td>0.24</td>
<td>0.24</td>
<td>0.48</td>
<td>0.26</td>
</tr>
<tr>
<td>3</td>
<td>0.16</td>
<td>0.17</td>
<td>0.27</td>
<td>0.14</td>
</tr>
<tr>
<td>4</td>
<td>0.09</td>
<td>0.11</td>
<td>0.20</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>0.07</td>
<td>0.08</td>
<td>0.14</td>
<td>0.07</td>
</tr>
<tr>
<td>6</td>
<td>0.06</td>
<td>0.07</td>
<td>0.12</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 8: Shift1D-A2: a(i) = B(i+10).

For a shift of ten elements like in assignment A2, together with a cyclic distribution onto two or five processors, no communication is needed because all elements are owned by the same proces-
**EXPERIMENTAL RESULTS**

... (marked with ellipses in Table 8). Both the PRE-HPF and the PGI-HPF compiler could, of course, not recognize that because the number of processors was not known at compile-time.

Although, the PRE-HPF compiler did not detect this fact, it still generates code that performs as good as a detected local assignment (same timings as in a block distribution). Whereas the program generated by the PGI-HPF compiler takes about 20.55 seconds and 8.67 seconds respectively, the PRE-HPF generated program takes about .23 and 0.06 seconds, a factor between 90 and 144!

Although it might seem far-fetched to concentrate on these two special communication free cases, HPF programmers do expect to see the timings drop considerably when no communication is needed. After all, that is why they are willing to spend so much effort in proper aligning arrays.

### 8.2.5 Stencil assignments

The main reason why we made a distinction between shifting an array and a stencil assignment as shown in Example 64, is that the PRE-HPF compiler treats them differently. For this assignment the PRE-HPF compiler does use the stencil optimization as described in Section 6.1.4, albeit only for block distributions and stencil offsets less than the default shadow width $C\_ovl$. The default shadow width for the PRE-HPF compiler is set to five. A larger value would mean allocating more memory that might not be used. A smaller value reduces the chances to optimize stencil references.

```fortran
PROGRAM stencil1D
  INTEGER, parameter :: N=60000, imaxa=100, M=N/(4*NUMBER_OF_PROCESSORS())
  INTEGER :: i, iter, l, u, s
  REAL :: A(N), B(N), tim1, tim2, time_us

  !HPFS PROCESSORS P(NUMBER_OF_PROCESSORS())
  !D1!!HPFS DISTRIBUTED (BLOCK ) ONTO P :: A, B
  !D2!!HPFS DISTRIBUTED (CYCLIC ) ONTO P :: A, B
  !D3!!HPFS DISTRIBUTED (CYCLIC(M)) ONTO P :: A, B
  !D4!!HPFS DISTRIBUTED (CYCLIC(4)) ONTO P :: A, B
  tim1 = time_us()
  DO iter = 1, imaxa
    CALL values(N, iter, l, u, s)
    FORALL (i=l:u:s)
      !A1!
      A(i) = B(i) + B(i+1)
      !A2!
      A(i) = B(i) + B(i+10)
    END FORALL
  END DO
  tim2 = time_us()
  print tim2-tim1
END PROGRAM
```

Example 64: Stencil1D.fml
The two assignments in Example 65 are also treated differently by the PRE-HPF compiler. The first assignment $A1$ allows a stencil optimization for the block distribution, because the stencil offset does not exceed the default shadow width $C_{owl}$. Internally the single FORALL is split up into two normalized volume assignments (forall's) as shown in Example 65.

\begin{verbatim}
A,B : mapped (i) on P(block(i,m))

FORALL (i = -)
A1:  A(i) = B(i) + B(i+1)
END FORALL

⇒

A2t: forall (i = ... on P(i),P(i+1)) ! neighboring stencil assignment
    B(i+1) = on P(i+1){B(i+1)}
    end forall

A2: forall (i = ... on P(i)) ! local assignment
    A(i) = B(i) + B(i+1)
    end forall
\end{verbatim}

Example 65: Stencil optimization (stencil offset <= 5).

The first forall is a neighboring stencil assignment which fills the shadow areas of $B$ on processor $P(i)$. The second forall will do the actual assignment and uses the local and (originally) remote values which are now stored in the shadow area.

If the stencil offset exceeds the default shadow width the PRE-HPF compiler has to fall back onto copying the remote values into a temporary variable as shown in Example 66. The HPF compiler as described in this thesis would however detect it as a two owned assignment in which communication and computation can be mixed.
Example 66: No stencil optimization (stencil offset > 5).

The timings of the two assignments are given in Table 9 and Table 10.

### Table 9: Stencil1D-A1: \( a(i) = b(i) + b(i+1) \)

<table>
<thead>
<tr>
<th>np</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.58</td>
<td>0.64</td>
<td>0.56</td>
<td>1.18</td>
<td>8.10</td>
<td>2.29</td>
<td>1.46</td>
<td>1.16</td>
<td>1.46</td>
</tr>
<tr>
<td>2</td>
<td>0.28</td>
<td>0.29</td>
<td>0.36</td>
<td>0.50</td>
<td>7.45</td>
<td>14.49</td>
<td>0.78</td>
<td>28.11</td>
<td>1.41</td>
</tr>
<tr>
<td>3</td>
<td>0.16</td>
<td>0.18</td>
<td>0.52</td>
<td>0.29</td>
<td>7.22</td>
<td>10.13</td>
<td>0.46</td>
<td>22.06</td>
<td>0.67</td>
</tr>
<tr>
<td>4</td>
<td>0.11</td>
<td>0.13</td>
<td>0.14</td>
<td>0.20</td>
<td>7.19</td>
<td>6.79</td>
<td>0.31</td>
<td>14.44</td>
<td>0.58</td>
</tr>
<tr>
<td>5</td>
<td>0.09</td>
<td>0.09</td>
<td>0.11</td>
<td>0.14</td>
<td>7.09</td>
<td>5.62</td>
<td>0.24</td>
<td>12.49</td>
<td>0.46</td>
</tr>
<tr>
<td>6</td>
<td>0.08</td>
<td>0.08</td>
<td>0.09</td>
<td>0.13</td>
<td>7.07</td>
<td>4.33</td>
<td>0.21</td>
<td>9.72</td>
<td>0.46</td>
</tr>
</tbody>
</table>

### Table 10: Stencil1D-A2: \( a(i) = b(i) + b(i+10) \)

<table>
<thead>
<tr>
<th>np</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
<th>pg</th>
<th>pre</th>
<th>gmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.17</td>
<td>1.49</td>
<td>1.88</td>
<td>1.17</td>
<td>8.12</td>
<td>2.31</td>
<td>1.46</td>
<td>1.16</td>
<td>1.48</td>
<td>20.31</td>
<td>0.60</td>
<td>30.91</td>
</tr>
<tr>
<td>2</td>
<td>0.43</td>
<td>0.63</td>
<td>0.95</td>
<td>0.56</td>
<td>7.51</td>
<td>15.28</td>
<td>0.99</td>
<td>20.97</td>
<td>0.60</td>
<td>30.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.33</td>
<td>0.38</td>
<td>0.57</td>
<td>0.26</td>
<td>7.27</td>
<td>9.05</td>
<td>0.56</td>
<td>22.31</td>
<td>0.94</td>
<td>30.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.20</td>
<td>0.26</td>
<td>0.36</td>
<td>0.29</td>
<td>7.19</td>
<td>8.09</td>
<td>0.57</td>
<td>14.33</td>
<td>0.55</td>
<td>20.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.29</td>
<td>0.22</td>
<td>0.29</td>
<td>0.16</td>
<td>7.14</td>
<td>6.59</td>
<td>0.43</td>
<td>8.84</td>
<td>0.16</td>
<td>17.69</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.24</td>
<td>0.58</td>
<td>0.20</td>
<td>0.18</td>
<td>7.12</td>
<td>7.10</td>
<td>0.34</td>
<td>9.04</td>
<td>0.31</td>
<td>19.21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The big difference occurs for the **PRE-HPF** compiler for the block distribution. That is because, in assignment A1 only the remote values are assigned, while in assignment A2 all values, remote and local, are assigned.

### 8.2.6 Sender Computes Rule

The **PRE-HPF** compiler can also detect that, when the complete rhs is owned by a single owner, the **Sender Computes Rule (SCR)** can be used, as given in assignment Example 67. The sub-expression $B\text{(i+10)} + D\text{(i+10)}$ is executed on the sending processor. The **PGI-HPF** compiler uses the **Owner Computes Rule (OCR)** and needs to introduce two additional copies aligned with the lhs, resulting into two communication assignments and one local assignment.

```fortran
PROGRAM SCR1D
  INTEGER, parameter :: N=60000, imaxa=100, M=N/(4*NUMBER_OF_PROCESSORS())
  INTEGER :: i, iter, l, u, s
  REAL :: A(N), B(N), D(N), tim1, tim2, time_us
!HPFS PROCESSORS P(NUMBER_OF_PROCESSORS())
!D1!!HPFS DISTRIBUTED (BLOCK ) ONTO P :: A,B,D
!D2!!HPFS DISTRIBUTED (CYCLIC ) ONTO P :: A,B,D
!D3!!HPFS DISTRIBUTED (CYCLIC(N)) ONTO P :: A,B,D
!D4!!HPFS DISTRIBUTED (CYCLIC(4)) ONTO P :: A,B,D
  tim1 = time_us()
  DO iter = 1, imaxa
    CALL values(N, iter, l, u, s)
    DOALL (i=1:u:s)
      A(i) = B(i+10) + D(i+10)
    END DOALL
  END DO
  tim2 = time_us()
  print tim2-tim1
END PROGRAM
```

**Example 67: Scr1D.f90**

As can be seen in Table 11 the **PRE-HPF** compiler outperforms the **PGI-HPF** compiler for most of the distributions. Except of course for the block-like distribution where the inefficient enumeration method when packing, again, dominates all the **PRE-HPF** timings.
a(i) = b(i+10) + d(i+10)

<table>
<thead>
<tr>
<th>np</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.76</td>
<td>0.60</td>
<td>2.76</td>
<td>1.77</td>
<td>5.95</td>
<td>2.89</td>
<td>0.80</td>
<td>1.80</td>
<td>0.80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.05</td>
<td>0.48</td>
<td>1.33</td>
<td>1.09</td>
<td>6.52</td>
<td>19.88</td>
<td>0.82</td>
<td>25.24</td>
<td>0.46</td>
<td>44.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.72</td>
<td>0.29</td>
<td>0.81</td>
<td>0.71</td>
<td>6.74</td>
<td>11.95</td>
<td>0.49</td>
<td>32.76</td>
<td>0.85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.57</td>
<td>0.18</td>
<td>0.54</td>
<td>0.51</td>
<td>6.67</td>
<td>11.62</td>
<td>0.59</td>
<td>19.84</td>
<td>0.50</td>
<td>31.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.44</td>
<td>0.14</td>
<td>0.42</td>
<td>0.98</td>
<td>6.71</td>
<td>9.62</td>
<td>0.40</td>
<td>10.42</td>
<td>0.08</td>
<td>28.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.39</td>
<td>0.15</td>
<td>0.28</td>
<td>0.33</td>
<td>6.75</td>
<td>11.11</td>
<td>0.32</td>
<td>11.91</td>
<td>0.31</td>
<td>33.02</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Sender Computes rule: A(i) = B(i+10) + D(i+10).

For the PRE-HPF compiler the assignment $A(i) = B(i+10) + D(i+10)$ is treated almost the same as $A(i) = B(i+10)$, except for the extra addition that is needed at the sending processor. When we compare Table 11 with Table 8, we see only a slight increase of PRE-HPF timings because of this extra addition. The PGI-HPF timings however increase considerably. This is directly related to the fact that the PRE-HPF compiler can avoid an extra communication statement.

As the Sender Computes Rule is only a rudimentary approach compared to our alignment analysis, we expect to see a much better performance for our HPF compiler.

**8.2.7 Semi-regular assignments**

Four assignments are used to test the efficiency of irregular communication patterns as shown in Example 68.
PROGRAM semireg1D

INTEGER, parameter :: N=60000, imaxa=10, M=N/(4*NUMBER_OF_PROCESSORS())
INTEGER :: i, iter, l, u, s
REAL :: A(N), B(N), ind(N), tim1, tim2, time_us

! HPFS PROCESSORS P( NUMBER_OF_PROCESSORS())
! D1!!HPFS$ DISTRIBUTE (BLOCK ) ONTO P :: A, B
! D2!!HPFS$ DISTRIBUTE (CYCLIC ) ONTO P :: A, B
! D3!!HPFS$ DISTRIBUTE (CYCLIC(M)) ONTO P :: A, B
! D4!!HPFS$ DISTRIBUTE (CYCLIC(4)) ONTO P :: A, B

! A1,3! FORALL (i=1:N) ind(i) = i
! A2,4! FORALL (i=1:N) ind(i) = N - i + 1

tim1 = time_us()
DO iter = 1, imaxa
    CALL values(N, iter, l, u, s)
    FORALL (i=1:u:s)
        ! A1,2! A(i) = B(ind(i))
        ! A3,4! A(ind(i)) = B(i)
    END FORALL
END DO

tim2 = time_us()
print tim2-tim1
END PROGRAM

---

Example 68: Semireg1D.tmpl

The first two are semi-regular gather assignments. The last two are semi-regular scatter assignments. The assignments A1 and A3 have such an indirection array that no actual communication is needed. The iteration space however, needs to be scanned completely just to find out there is no communication. The gather and scatter assignments will be discussed below in separate subsections because they are treated differently by the PRE-HPF compiler.

Semi-regular: Gather

For the first two assignments, A1 and A2, the PRE-HPF compiler will use an inspector/executor approach (a derivative of the PARTI [69] library package). The reason for this is as follows. Because the PRE-HPF compiler sends messages as soon as they are packed, it needs to find an upperbound on the size of the send buffer for each processor. In the case of a gather assignment, a proper upper bound for the send buffer can not be determined. Therefore, it has to fall back onto the more expensive inspector/executor approach.

Our HPF compiler, however, is designed to first pack all messages into one send buffer for which even in gather assignments a proper upperbound on its size can be determined. The number of elements that possibly need to be communicated on a certain processor $p$ can never exceed the number of local elements of $B$ on processor $p$, provided no duplicates are send.

The timings of assignment A1 can be found in Table 12, those of assignment A2 can be found in Table 13.
\( a(i) = b(\text{ind}(i)), \quad \text{ind}(i) = i \)

<table>
<thead>
<tr>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>pg</td>
<td>pre</td>
<td>gmd</td>
</tr>
<tr>
<td>1</td>
<td>0.70</td>
<td>0.80</td>
<td>3.70</td>
</tr>
<tr>
<td>2</td>
<td>0.64</td>
<td>0.40</td>
<td>3.61</td>
</tr>
<tr>
<td>3</td>
<td>0.44</td>
<td>0.30</td>
<td>3.57</td>
</tr>
<tr>
<td>4</td>
<td>0.37</td>
<td>0.25</td>
<td>3.55</td>
</tr>
<tr>
<td>5</td>
<td>0.27</td>
<td>0.22</td>
<td>3.55</td>
</tr>
<tr>
<td>6</td>
<td>0.22</td>
<td>0.20</td>
<td>3.54</td>
</tr>
</tbody>
</table>

Table 12: Semireg-A1: Gather without communication

The PRE-HPF compiler slightly outperforms the PGI-HPF compiler for the block distribution. For the block-like and cyclic-like distributions, the performance figures of the PRE-HPF compiler and the PGI-HPF compiler are more or less comparable, because both compilers rely on the same kind of enumeration. For the cyclic distribution, the PRE-HPF compiler performs best.

The GMD-HPF compiler does not scale at all, because it probably falls back onto element-wise communication. Although there is no communication.

\( a(i) = b(\text{ind}(i)), \quad \text{ind}(i) = n - i \)

<table>
<thead>
<tr>
<th>block</th>
<th>block-like</th>
<th>cyclic-like</th>
<th>cyclic</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>pg</td>
<td>pre</td>
<td>gmd</td>
</tr>
<tr>
<td>1</td>
<td>0.70</td>
<td>0.80</td>
<td>3.70</td>
</tr>
<tr>
<td>2</td>
<td>0.82</td>
<td>1.01</td>
<td>14.91</td>
</tr>
<tr>
<td>3</td>
<td>0.56</td>
<td>0.70</td>
<td>11.13</td>
</tr>
<tr>
<td>4</td>
<td>0.41</td>
<td>0.54</td>
<td>9.23</td>
</tr>
<tr>
<td>5</td>
<td>0.33</td>
<td>0.47</td>
<td>8.09</td>
</tr>
<tr>
<td>6</td>
<td>0.28</td>
<td>0.39</td>
<td>7.30</td>
</tr>
</tbody>
</table>

Table 13: Semireg-A1: Gather with communication

Adding communication will cause the timings of the PRE-HPF compiler to increase considerably. The advantage for the block distribution is lost and for the block-like and cyclic-like distributions, the PGI-HPF compiler is definitely better. Only for the cyclic distribution, the PRE-HPF compiler is better. The main reason why the PRE-HPF compiler stays behind for the cyclic-like distribution is that the PRE-HPF compiler uses a rather inefficient communication schedule as mention before in section 8.2.3. Switching to scheduling at run-time would probably result in better scaling of execution times. The GMD-HPF compiler cannot match up with the other two compilers.
Semi-regular: Scatter

The semantics of the FORALL prescribe that different elements must be assigned. Therefore, there are no duplicate values in the indirection array and a proper upperbound can be found for the size of the receive buffer on each processor. This allows the semi-regular assignment approach to be used for the last two assignments, A3 and A4 as described in Section 6.1.7.

The PRE-HPF compiler performs for the scatter assignment much better as compared to the gather assignment, as can be seen in Table 14 and Table 15.

<table>
<thead>
<tr>
<th>np</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.70</td>
<td>0.17</td>
<td>3.65</td>
<td>0.94</td>
<td>0.52</td>
<td></td>
<td></td>
<td>1.08</td>
<td>0.35</td>
<td>4.28</td>
<td>1.23</td>
<td>0.24</td>
<td>3.87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>0.21</td>
<td>3.56</td>
<td>0.90</td>
<td>0.69</td>
<td></td>
<td></td>
<td>1.13</td>
<td>0.48</td>
<td>6.46</td>
<td>1.79</td>
<td>0.30</td>
<td>5.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.61</td>
<td>0.22</td>
<td>3.55</td>
<td>0.67</td>
<td>0.75</td>
<td></td>
<td></td>
<td>0.92</td>
<td>0.51</td>
<td>6.40</td>
<td>1.82</td>
<td>0.33</td>
<td>5.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.45</td>
<td>0.21</td>
<td>3.53</td>
<td>0.84</td>
<td>0.78</td>
<td></td>
<td></td>
<td>1.42</td>
<td>0.54</td>
<td>6.42</td>
<td>1.04</td>
<td>0.32</td>
<td>5.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.46</td>
<td>0.21</td>
<td>3.55</td>
<td>0.75</td>
<td>0.80</td>
<td></td>
<td></td>
<td>0.76</td>
<td>0.56</td>
<td>6.39</td>
<td>0.97</td>
<td>0.33</td>
<td>5.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.71</td>
<td>0.22</td>
<td>3.50</td>
<td>0.80</td>
<td>0.81</td>
<td></td>
<td></td>
<td>0.50</td>
<td>0.56</td>
<td>6.38</td>
<td>0.91</td>
<td>0.34</td>
<td>5.88</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 14: Semireg-A3: Scatter without communication

The timings of PRE-HPF compiler are much better then the timings of the PGI-HPF compiler, however they do not scale. That last remark also holds also for the timings of the PGI-HPF and GMD-HPF compiler. In case of the PRE-HPF compiler, the timings do not scale because the amount of work needed to pack and unpack the messages does not depend on the number of processors used. The processor-scanning loop will increase and the scanned iteration space per processor will decrease, but the total amount scanned on a single processor is equal to the iteration space. For the block-like distribution, both compilers perform equally well, or worse.

<table>
<thead>
<tr>
<th>np</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
<th>pgi</th>
<th>pre</th>
<th>gmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.70</td>
<td>0.17</td>
<td>3.67</td>
<td>0.94</td>
<td>0.52</td>
<td></td>
<td></td>
<td>1.07</td>
<td>0.34</td>
<td>4.29</td>
<td>1.24</td>
<td>0.24</td>
<td>3.80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.11</td>
<td>0.29</td>
<td>14.89</td>
<td>1.07</td>
<td>0.76</td>
<td></td>
<td></td>
<td>1.15</td>
<td>0.54</td>
<td>19.85</td>
<td>1.88</td>
<td>0.40</td>
<td>28.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.84</td>
<td>0.35</td>
<td>11.26</td>
<td>0.77</td>
<td>0.79</td>
<td></td>
<td></td>
<td>0.81</td>
<td>0.56</td>
<td>15.34</td>
<td>1.31</td>
<td>0.44</td>
<td>20.80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.83</td>
<td>0.24</td>
<td>9.23</td>
<td>0.61</td>
<td>0.80</td>
<td></td>
<td></td>
<td>0.69</td>
<td>0.57</td>
<td>12.98</td>
<td>1.17</td>
<td>0.38</td>
<td>17.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.59</td>
<td>0.25</td>
<td>8.05</td>
<td>0.51</td>
<td>0.82</td>
<td></td>
<td></td>
<td>0.84</td>
<td>0.62</td>
<td>11.61</td>
<td>0.82</td>
<td>0.36</td>
<td>15.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.38</td>
<td>0.26</td>
<td>7.28</td>
<td>0.48</td>
<td>0.82</td>
<td></td>
<td></td>
<td>0.64</td>
<td>0.60</td>
<td>10.65</td>
<td>0.73</td>
<td>0.37</td>
<td>13.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 15: Semireg-A4: Scatter with communication
Adding communication only increases the timings slightly for the PRE-HPF compiler. For the PGI-HPF compiler the timings can either increase or even decrease! The GMD-HPF compiler timings are again no match for the other two compilers.

8.2.8 Conclusions

The main conclusions that can be drawn from the TNO benchmarks is that our HPF compiler must at least, as designed, generate code in which:

- Row-wise and column-wise enumeration can be selected at run-time.
- All packing has finished before sending messages. In that case, an optimal scheduling of messages can be found at run-time.
- The sender and receiver computes rule is implemented.

Unfortunately, the PRE-HPF compiler does not support the above features and the HPF compiler as described in this thesis is still a design. Therefore, the full potential of our HPF compiler cannot be shown by just comparing the PRE-HPF compiler with the PGI-HPF compiler. Nevertheless, because the PRE-HPF compiler does already compare nicely with the PGI-HPF compiler, we expect that our HPF compiler will perform much better.

8.3 Kernel TNO-Wish3d

To evaluate the performance of a real application we used an existing Computation Fluid Dynamics (CFD) program called TNO-Wish3d. TNO-Wish3d is an in-house developed CFD-model capable of calculating the steady-state or time-dependent three-dimensional temperature, velocity, turbulence quantities and species concentration distributions in combustion chambers of glass melting furnaces. This module is used to simulate the (combustion) space above the surface of the glass melt of industrial float, container, table-ware, fibre, TV-panel, funnel and E-glass furnaces.

The heat transfer by both radiation and turbulent convection to the glass can be calculated in the melter, (re)finer, neck, waist, working end and throat can be calculated. The furnace can contain gas- and oil-fired burners, air- and oxy-firing as well as oxy-boosting or oxygen enrichment. The model is able to simulate different geometries and process conditions and can be used to optimize furnace design in order to obtain an optimal glass melting process, investigate effects of burner type, improve glass quality, improve energy efficiency, reduce emissions of pollutants such as NOx, and maximize furnace life-time by reducing condensation on and oxidation of refractories. The structure of the CFD-model TNO-Wish3d is as follows:
Because of the size of TNO-Wish3d, which is more than 100,000 lines of source code and the fact that our PRE-HPF compiler did not support the complete HPF library we had to constrain ourselves to a kernel code. The kernel was selected in such a way that it contained code that is typically used in full CFD programs that consumed most of the computation time. It turned out to be a solver algorithm as given in Example 69.

The subroutine linx solves a linear differential equation in three dimensions. Because there is a loop carried dependency in the x-direction, the i-index is enumerated in a given order with a do loop. The other two dimensions, which do not contain loop carried dependencies, are enumerated with a FORALL loop. The dummy parameter var as well as the local variables are all aligned to a three dimensional template T. Therefore, it is quite easy for the compiler to find out the owners of each of the array references in the FORALL loops.
SUBROUTINE linx( var, relvar, ibeg, iend,
& mjend, jbeg, jend, jstp,
& mkend, kbeg, kend, kstp)
INTEGER ibeg, iend, mjend, jbeg, jend, jstp, mkend, kbeg, kend, kstp
DOUBLE PRECISION var(ndx, ndy, ndz), relvar
INTEGER i, j, k, m
DOUBLE PRECISION dimension(ndx, ndy, ndz) :: beta, delt, varn, term

!HPF$ ALIGN WITH T:: var
!HPF$ ALIGN WITH T:: beta,delt,varn,term

DO m = 1, mjend * mkend
  DO (k = kbeg:kend, j = jbeg:jend)
    delt(ibeg-1,j,k) = 0.
    betam(i,j,k) = var(ibeg-1,j,k)
  END DO
  DO (i = ibeg:iend)
    betam(i,j,k) = aem(i,j,k)
    delt(i,j,k) = asm(i,j,k) * var(i,j-1,k) +
    & anm(i,j,k) * var(i,j+1,k) +
    & abm(i,j,k) * var(i,j,k-1) +
    & atm(i,j,k) * var(i,j,k+1) + scm(i,j,k)
  END DO
  DO (i = ibeg, iend)
    DO (k = kbeg:kend, j = jbeg:jend)
      term(i,j,k) = 1. / (apm(i,j,k) - awm(i,j,k) * beta(i-1,j,k))
      betam(i,j,k) = betam(i,j,k) * term(i,j,k)
      delt(i,j,k) = (deltm(i,j,k) + awm(i,j,k) * delt(i-1,j,k)) * term(i,j,k)
    END DO
  END DO
IF (relvar .gt. 0.) then
  DO (i = iend, ibeg, -1)
    DO (k = kbeg:kend, j = jbeg:jend)
      var(i,j,k) = betam(i,j,k) * var(i+1,j,k) + delt(i,j,k)
    END DO
  END DO
ELSE
  DO (k = kbeg:kend, j = jbeg:jend)
    varn(iend+1,j,k) = var(iend+1,j,k)
  END DO
  DO (i = iend, ibeg, -1)
    DO (k = kbeg:kend, j = jbeg:jend)
      var(i,j,k) = betam(i,j,k) * var(i+1,j,k) + delt(i,j,k)
    END DO
  END DO
END IF
END DO
END SUBROUTINE

Example 68: The TNO-Wish3d Kernel code
The subroutine \textit{lin}x is called together with its sister subroutines \textit{lin}y and \textit{lin}z, in a time-step loop. The idea is that for each step in the time-step loop the result will converge to a final result that is closer to the solution of given partial differential equations, describing the problem of interest.

When running the kernel code the elapsed wall clock time at processor 0 is measured and treated as the elapsed time of the program. Because processor 0 handles all I/O it will be the first process that starts and the last process that ends processing, and therefore it is a good estimation of the elapsed time of the complete parallel program. Starting and stopping the clock is done inside the main program via a subroutine call, so it excludes the startup and cleanup time of the \( n \) processes running on the parallel system.

Measurements of the kernel were done on different machines. We used a Cray T3E (up to 8 nodes), and the DAS machine. For both machines, a commercial compiler is also used to compare with our PRE-HPF compiler. Because PGI uses different version numbers for different machines, both PGI compilers can not be compared directly. The version of the Cray PGI-HPF compiler, however, was released several years earlier compared to the one we used on the DAS machine. The results of the measurements are given in Figure 29.

![Graph](image-url)

\textbf{Figure 29: Timings of Kernel TNO-Wish3d, for a (block, \*, \*) distributed template.}

Some conclusions can be drawn from these figures.
• All compilers generate code that scales on different types of machines, except for the older PGI-HPF compiler on the Cray-T3E. This can be seen by the shape of the curves, which behave like 1/p, where p is the number of running processors. This illustrates one of the design criteria of HPF: “Portability of efficiency without relying onto machine specific source code”.

• On the DAS machine, the PRE-HPF compiler outperforms the PGI-HPF compiler with 20 percent. This is not bad considering that the PRE-HPF compiler is a non-tuned prototype. For example the dummy variables var of the subroutines linx, liny, and linz, are copied in and out by the calling subroutine for each invocation of the called subroutine. In a fine-tuned compiler, only the mapping descriptor is passed on to the called subroutine.

• On the Cray-T3E, the performance of the PRE-HPF compiler compared with the PGI-HPF compiler is very good. In fact, the commercial compiler performs badly because it does not even scale. One of the reasons is that the old PGI-HPF compiler performs so badly is that it is actually based on a shared-memory model. That’s is fine when working on a shared memory like the Cray T3D but not on a distributed-memory machine like the DAS machine or the Cray T3E. We expect however that this performance bug will be removed in a later version of the HPF compiler on the Cray T3E. After all, they already solved it for the DAS machine.

• Given the same program and the same PRE-HPF compiler the performance of the Distributed ASCII Supercomputer (DAS) comes very close to the performance of the Cray T3E dedicated supercomputer. Given the fact that the DAS machine only costs a fraction of the Cray T3E it has a price performance ratio that can not be beaten by dedicated supercomputers.

8.4 Comparing overhead of different enumeration/storage methods

8.4.1 Index Generation Overhead

The results that are shown here are the execution times of hand-written code that mimics the behavior of compiler-generated code. We use this code instead of real code because it makes it simpler to evaluate variants on the implementation, to simulate implementation schemes proposed by others, and to ensure the accuracy of the measurements.

We have made a comparison between our local enumeration and storage methods and the pattern cyclic method of Chatterjee et al [18] with respect to the overhead generated to access local elements. The code simulates the execution of the statement (see Example 70) using a one-dimensional distributed array of size 40000.
Example 70: Test assignment for index generation overhead

We assume that the address calculation for $A(a*i+b)$ is done only once, so that in effect this statement is an increment operation. This statement was chosen because it makes it simple to verify that all elements of the array have been enumerated exactly once, independently of the order in which they have been enumerated.

We assume that rowwise/columnwise enumeration is aligned with rowwise/columnwise storage, thereby showing the same execution times. In theory the method we simulate, columnwise enumeration, should perform slightly worse than rowwise enumeration, because it requires an additional gcd calculation. In practice, the results for both methods showed little difference. We further assume that pattern cyclic enumeration method adds an additional array reference for the look-up table as compared to the methods described in this paper. This is more efficient than the original implementation of Chatterjee et al., and corresponds to a more recent implementation described by Kennedy et al. [40] (method C on page 13 in their paper). We ignore the time that is required for table construction.

For all methods, three levels of optimization were assumed. In the slowest variant, the global-to-local calculation is done for every element that is enumerated. In the next variant, only one global-to-local calculation is done for each row, column or pattern-cycle. In the last variant, all global-to-local calculations have been eliminated by using a table of pre-computed values.

We assume that the two optimizations can also be used in the pattern-cyclic method. Finally, the sequential version of the statement has also been included to serve as a comparison.
The experiment was done on a number of different processors: a microSPARC II and other SPARC variants, a Hewlett Packard PA-RISC 1.1, and a MIPS R4400. For the PA-RISC machine the native Hewlett Packard C compiler was used with option -O, in all other cases the GNU C compiler Version 2.7.2 was used with option -O2. Essentially, all experiments yielded similar results.

A typical result is depicted in Figure 30 (log-log scale); this is for a microSPARC II on 85 MHz. The label 'row or column' stands for the enumeration method described in this paper. The variants 'without g2l' assume that only one global-to-local calculation is necessary for each row, column, or pattern cycle. In the variant 'with table', the starting points of the rows or columns are pre-computed.

Several observations can be made from this figure:

- For linear access sequences and large inner loops, the pattern-cyclic method is about 50% slower than the method described in this paper. Measurements on other machines yielded similar results (50% to 100% slower).

- On short inner loops (<50) all methods give an increased overhead.
• For inner loops larger than 100, the loop overhead is comparable to pure sequential execution.

• Full global-to-local conversions for arbitrary array accesses are in all methods expensive (up to about 40 times the sequential execution).

Figure 31: The execution time of the enumeration methods for a number of distributions.

However, there is more to be said. The size of the inner loop can not be freely chosen. In both the pattern cyclic and the rowwise enumeration method, the size of the inner loop is bounded by $m$.

We have taken three different values of $m; m = 4, m = 40,$ and $m = 400$ and marked the outcome of each enumeration method on the curves with only global-to-local calculations in the outer loop. This is shown in Figure 31. Point $s$ shows the execution time for sequential execution. The points $r1, r2, r3$ show execution times for rowwise enumeration; points $c1, c2, c3$ show execution times for columnwise enumeration; and points $p1, p2, p3$ show execution times for pattern cyclic
enumeration. The points \( r_1, c_1, p_1 \) are for an array distribution with \( m = 4 \). The points \( r_2, c_2, p_2 \) are for an array distribution with \( m = 40 \), and the points \( r_3, c_3, p_3 \) are for an array distribution with \( m = 400 \).

As is obvious from the graph, no method is best for all distributions we considered: for \( m=4 \) and \( m=40 \), columnwise enumeration is best, and for \( m=400 \) rowwise enumeration is best. In general, for small values of \( m \), columnwise iteration will generally produce longer vectors and hence less overhead. For larger values of \( m \), rowwise and pattern-cyclic enumeration becomes more efficient. Pattern-cyclic enumeration can often span larger vectors than rowwise enumeration, but incurs more overheads.

### 8.4.2 Storage Overhead

So far, we only concentrated on the performance of the generated program, but what about the memory requirements of the parallel program. To get a truly memory-scaleable parallel program the overall memory requirements may not depend on the number of processors. In two places in the compiler design, memory requirements will show up. First when allocating mapped variables, and secondly, when executing two-owned assignments as send and receive buffers.

For the allocation of mapped variables, we want to allocate as little additional memory as possible; to avoid allocating memory that is never referenced (the so called memory holes). We will allow some memory holes to ease the reference and enumeration of local elements. For instance, in the block_cyclic-distributed case, memory holes are introduced at allocation time to keep the local storage two-dimensional. That way, the local elements can easily be enumerated with two do loops and the local address pointer can be expressed as an induction variable of both loops.
As described in Section 5.3.1 and shown in Figure 32 storage compression can either be done row-wise or column-wise, each of them resulting in a different number of holes allocated. Because holes can only occur at the end of a compressed dimension, there is at most a fraction of $1/c'_{\text{ext}}$ for column-wise compression and $1/r'_{\text{ext}}$ holes for row-wise compression. $c'_{\text{ext}}$ and $r'_{\text{ext}}$ are the extents of the compressed column and row indices. The compiler may choose row-wise or column-wise storage compression depending on the smallest fraction\(^{52}\). In case both fractions are equal (i.e. a square 2 dimensional local memory) the storage overhead is worst case $1/\sqrt{N_{\text{loc}}}$, where $N_{\text{loc}}$ is the number of elements locally owned by a processor. Normally the number of elements is much larger than 100, otherwise it would not make sense to distribute the array. Therefore, the storage overhead is normally less than 10%.

\(^{52}\) As proven in [66] the storage compression can be selected independently of the usage of the array.
9 Conclusions

We have presented a compilation framework suitable for shared memory and distributed memory machines. It relies on a Parallel Abstract Machine for full portability. The Parallel Abstract Machine can be implemented on top of standard communication libraries like PVM or MPI. The compilation framework uses the Single Program Multiple Data as its basic execution model. The owner model allows deviating from the Single Program Multiple Data execution model whenever possible.

The compilation system is capable of handling all HPF data distributions and stores them internally as a small set of mapping attributes. Owner attributes allow the user to specify where parts of the program must be executed. Both data and work distributions can be mixed in a single program.

The compilation system is modeled as a set of engines working on the same internal representation. The align-engine, the decompose-engine, and the SPMD-engine, together the owner-aware engines, and can be mixed with 'classical' sequential optimizers, like constant folding, constant propagation, and invariant code motion.

The align-engine can symbolically derive mappings and owners for the unspecified parts of the program, thereby fixing the exact distribution of data and work for the complete program. A full search algorithm can be applied which finds the optimal alignments. A two-sweep algorithm is provided as a more efficient alternative. The cost function does not need the complete mapping at compile-time and can deal with e.g. inherited mappings.

The decompose-engine can decompose the attributed program into sequences of basic assignments. Irregular assignments are decomposed into semi-regular assignments. We have shown that multi-owned expressions can be decomposed into a sequence of single-owned and two-owned assignments. By using temporary variables with mapping derived from the context, communication can be avoided and scalability with respect to memory usage is preserved as far as possible.

The SPMD-engine can translate basic assignments into efficient SPMD code. The allowed basic assignments are much more expressive than commonly used in other compilation frameworks. Other compilation frameworks normally only allow communication between two array variables. The iteration space has to be rectangular like in an array assignment. An extra array variable is allowed for indirection on either the lhs or rhs of the assignment. Other compilation frameworks do the complete handling of packing, communicating, and assignment is normally in a run-time support library. Our compilation framework relies as much as possible onto inlining of code fragments. The code fragments are designed to be small to allow highly tuned code depending on its context.

Our basic assignment is much more general compared to other compilation systems. It allows non-rectangular iteration spaces. When lowered the packing and assignment phases of the generated code are completely inlined, to allow further 'sequential' optimizations to follow. It supports
computation before packing at the sending side and computation at the receiving side after assignment. Reducing assignments where the volume of assigned elements is less than the iteration volume can also be expressed as a basic assignment. Its only restriction is that the assignment is a single or two-owned assignment. Many temporary variables and unnecessary copying can be avoided when using this general basic assignment.

A perfect balance between compiler-known mapping information and run-time known mapping information is found. For instance, for the inlining of the packing and assignment phase, only the distribution type is needed at compile-time. All other owner information may only be known at run-time.

The basic technique for efficient packing and assigning is based on owner absorption, which can be applied repeatedly. This allows a great deal of local optimizations, which together can result in powerful global optimizations.

By giving examples, we have shown that the compiler can derive data-mappings for temporary variables by inspecting the type of the expression for which it needs to store the values. Hoisting a mapped variable out of a loop is still possible. Its mapping can be adjusted by accordingly with the most appropriate mapping giving its context. Because data-mappings are modeled as a type extension, proper mappings can be found for variables for which no mapping is specified, based on its usage.

The experimental results with the PRE-HPF compiler, a predecessor of the CoSy compiler described in this thesis, show excellent performance for the generation of local element sets and the packing and assignment of messages. The scheduling of messages as implemented in the PRE-HPF compiler is fixed and therefore not properly tuned. Our compilation model packs all data on each processor into consecutive memory locations for each destination and leaves the complete scheduling of messages over to the run-time system.

Unfortunately, the real potential of our basic assignment that can be handled by the SPMD-engine can not be demonstrated, as the PRE-HPF compiler has only a rudimentary form of the SPMD-engine implemented. For those parts already implemented in the PRE-HPF compiler it can match and beat a widely available commercial HPF compiler.

The model of owner attributes allows many more places in the program where these owner attributes make sense. Several language extensions are proposed to give the programmer more opportunities to precisely specify data and work distributions.

The main hurdle we are facing now is an incomplete implementation of the owner model in the CoSy compiler, and an HPF front-end that accepts and translates all HPF features. It has blocked a proper evaluation of the performance of our model to its full details.

Because the compilation model was not completely implemented, we have not been able to experiment with different communication schedules on the two-owned assignments, as opposed to
the hard-coded schedule of the **PRE-HPF** compiler. Adding a rule-based rewriting system to the compilation framework can greatly speed up experiments with different translation schemes.

### 9.1 Future research

Future research will concentrate on more elaborate alignment analysis by overruling user-defined mappings of local variables if needed, extending the scope of the owner selection process to a basic block graph and procedure body. We will also look at further refinements of the framework to efficiently handle reductions and independent do-loops.

The next major step will be to extend a Java front-end [59] with HPF-like data attributes. It will give a clear insight in whether the same compilation model can be reused at the intermediate level of the CoSy compiler.

The execution model will be a deviation from the SPMD model by removing the restriction that all processors are always involved in the calling of subroutines or functions. The subset of processors that call subroutines or functions can be specified by the user. The active processor set of the called subroutine does not have to be a subset of the active processor set of the caller as required in the HPF2 task extensions.
Appendix

The two appendices list a variety of relevant Web sites of commercial and research HPF compilers.

Commercial HPF compilers

- EXPERT HPF (ACE): http://www.ace.nl/
- HPF Mapper (EPC): http://www.epc.co.uk/
- HPFplus (NAS): http://www.nasoftware.co.uk/
- pghpf (PGI): http://www.pgroup.com/
- XHPF (APR): http://www.apri.com/

Research compilers

- Adaptor (GMD-SCAI): http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html
- EPPP (CRIM Montreal): http://www.CRIM.ca
- Fortran D (Rice University): http://www.cs.rice.edu/~dsystem/
- VFC (University of Vienna): http://www.par.univie.ac.at/hpf%2b/index.html
- Paradigm (University of Illinois at Urbana-Champaign): http://www.crhc.uiuc.edu/Paradigm/
- SHPF (University of Southampton and VCPC) http://www.vcpc.univie.ac.at/information/software/shpf/
Summary

Design of an HPF Compiler

(A compilation framework for a data-parallel language)

Because HPF is a rather young language there are currently only a few compilers supporting the language. These compilers are mainly focused on getting the complete HPF language compiled. However to build a compiler qualified as a HIGH PERFORMANCE Fortran compiler the generated code needs to be very efficient with respect to speed and memory usage. There may be no efficiency loss, compared to a Fortran 77 compiler, on pure Fortran 77 programs. In addition, the efficiency of a data parallel HPF program must be comparable to an equivalent program, which uses a standard message-passing library, like PVM or MPI.

This thesis deals with the efficient compilation of HPF programs with data distributions on a distributed-memory machine, because distributed-memory machines are envisioned to be the future low-cost parallel machines.

The following research question is posed in this thesis. Does there exist an HPF compiler design that:

- Allows processing of all allowed HPF distributions?
- Can handle both data and work distributions efficiently?
- Does support optimizations, in a flexible and orthogonal way?
- Can be implemented in a simple and robust way?
- Can be easily integrated within an existing sequential compiler?

This thesis describes a design for a High Performance Fortran compiler that fulfills the above criteria as much as possible. The sketched framework can also be applied to other procedural data-parallel languages.

The design describes a mapping model that elegantly integrates data and work distributions within a sequential compiler framework. It shows where the HPF language can be extended to get a more expressive handling of mapped tasks and data within the same mapping model. The broad range of optimizations possible with the underlying mapping model are given by example.

The research method can best be described as incremental refinement. Once the first design was completed, an implementation was made. Because an HPF compiler adhering to the HPF standard was required, we were forced to check the consistency of our design into the smallest details. Once implemented, we could measure the speed of the generated code and fine-tune the optimizations where needed. This resulted in a new design. The first design was a result of the PRE-
PARE [73] project. The second design occurred within the FUTURE [26] project. The third design is documented in this thesis. A third implementation did not make it because of non-technical reasons.

To get an idea of the performance of our compiler, some experiments were done. For the experiments we either used the code generated by the compiler based on the second design (the PRE-HPF compiler) or simulated the output according to the described design. The time measurements of these experiments were compared with a commercial compiler and a free-ware compiler. We have shown that a factor, ranging from 1.2 to 144, can be gained in speed, by concentrating on the essentials of a data-parallel compiler design; the efficient enumeration and storage of distributed elements.

The design described in this thesis can directly be used in currently existing HPF compilers and makes performance portable between different HPF compilers. This will increase the acceptance of HPF as a simple programming model for distributed systems.

Will Denissen.
Ontwerp van een HPF compiler

(Een compilatie raamwerk voor een data parallele taal)

Omdat HPF een vrij nieuwe programmeer taal is zijn er nog niet veel HPF compilers beschikbaar. De eerste compilers die commercieel beschikbaar kwamen concentreerden zich op het correct compileren van de complete HPF taal. Om echter een als HIGH PERFORMANCE gekwalificeerde Fortran compiler te kunnen leveren dient deze code te genereren die efficiënt omgaat met zowel snelheid als geheugengebraak. Er mag bijvoorbeeld geen efficiëntie verlies optreden t.o.v. een Fortran 77 compiler, voor zuiver Fortran 77 programma. Tevens moet de efficiëntie van een data parallel HPF programma vergelijkbaar zijn met een equivalent programma dat een standaard ‘message-passing library’ zoals PVM of MPI gebruikt.

Dit proefschrift behandeld het efficiënt compileren van een HPF programma met data distributies op een distributed-memory machine, omdat distributed-memory machines gezien worden als de toekomstige goedkope parallele machines.

Als onderzoeksvraag hebben we ons de volgende vraag gesteld. Bestaat er een compiler ontwerp:

- Waarin alle HPF distributies verwerkt kunnen worden?
- Dat zowel data als werk distributies efficiënt ondersteunt?
- Dat optimalisaties flexibel en onafhankelijk van elkaar ondersteunt?
- Dat simpel en robuust geïmplementeerd kan worden?
- Dat eenvoudig geïntegreerd kan worden binnen een bestaande sequentiële compiler?

Dit proefschrift beschrijft een ontwerp voor een High Performance Fortran compiler dat zoveel mogelijk voldoet aan de bovenstaande criteria. Het raamwerk dat geschetst wordt is echter ook bruikbaar voor andere procedureel gerichte data parallele talen.

Het ontwerp beschrijft een mapping model data en werk distributies elegant integreert binnen een sequentieel compiler raamwerk. Het toont aan waar de HPF programmeertaal uitgebreid kan worden zodat een expressievere behandeling van gedistribueerde taken en data gerealiseerd kan worden. Voorbeelden tonen het grote aantal van mogelijke optimalisaties die mogelijk zijn binnen dit mapping model.

De methode van onderzoek die gebruikt is, is het best te omschrijven als die van stapsgewijze verfijning. Nadat een eerste versie van een compiler ontwerp was gemaakt volgde de implementatie. Door een volwaardige HPF compiler als uitgangspunt te nemen wordt men gedwongen tot in de kleinste details de consistente van het ontwerp te checken. Met behulp van de implementatie kan men door het meten van doorloop tijd van de door de compiler gegenereerde code de meest succesvolle optimalisaties selecteren en verfijnen. Hieruit volgt dan weer een verbeterd

Om een idee te krijgen van de performance van deze compiler zijn er enkele experimenten gedaan. Voor de experimenten is de gegenereerde code van de compiler gebruikt; gebaseerd op het tweede design (de PRE-HPF compiler) of gesimuleerde output volgens het beschreven design. De tijdsmetingen van deze experimenten werden vergeleken met de doorlooptijden van code zoals die door de meest voorkomende commerciële compiler werd gegenereerd. We hebben aangetoond dat factoren (variërend van 1.2 tot 144) in snelheid gewonnen kunnen worden door te concentreren op het meest elementaire van een dataparallel compiler ontwerp; het efficiënt opzomen en opslaan van gedistribueerde elementen.

Het in dit proefschrift beschreven design kan direct gebruikt worden in bestaande HPF compilers, waardoor prestaties overdraagbaar worden tussen verschillende HPF compilers. Dit alles zal de acceptatie van HPF als eenvoudig programmeermodel voor gedistribueerde systemen aanzienlijk vergroten.

Will Denissen.
Curriculum Vitae

Will Denissen was born in Berkel-Enschot on March 31st, 1962. After graduating from the “Cobbenhage College” in 1979 in Tilburg, he started at the “Hogere Technische School HTS” in Den Bosch to become an Electrical Engineer. After one year, he got the opportunity to start at the Eindhoven University of Technology. He received his MSc. degree in electrical engineering from the Eindhoven University of Technology in 1987. After working for a couple of years in two small companies, he currently works at the institute of applied physics of the Netherlands Organization for Applied Scientific Research (TNO-TPD).
Bibliography


[22] DAS: "The Distributed ASCI Supercomputer", URL: www.cs.vu.nl/das


Index

A

active processor set · 21; 44
ALIGN directive · 19
align-engine · 37
alignment function · 26
array assignment · 22
array descriptor · 38
attribute · 25; 36

B

basic assignment · 33
basic block · 32; 88
BEG · 35
\textit{BLOCK(M)} distribution · 17
Booster · 14; 83

C

cache coherency protocols · 7
cache memory · 7
CFD · 110
clusters · 7
CM-Fortran · 14
collapsed dimension · 17; 27
column-wise compression · 72
column-wise enumeration · 78
communication bandwidth · 9
COMPARE · 2
compilation system · 2
compiler domain · 26
Computational Fluid Dynamics · 110
conformable · 22
CoSy · 10; 35; 37
Cray-T3E · 8
cycle length · 70
\textit{CYCLIC(M)} distribution · 17

D

das · 116
data directives · 25
data parallel language · 1
decompose-engine · 37
dimension of an owner · 33
diophantine equations · 77
directive · 15; 25
\textit{DISTRIBUTE} directive · 17
Distributed ASCI Supercomputer · 116
Distributed Fortran · 14
distributed memory machine · 8
distributed-shared memory · 13
distribution function · 26
distribution type · 26
DO loop · 23
do-loopnest · 30
DSM · 13
dynamic type · 28

E

element replications · 57
engine · 35
enumeration equation · 77
execution model · 7; 11
executor phase · 97
explicit data distributions · 28
expression · 32
extended forall · 93
extent · 15

F

fetch before store · 22
fixed owner · 50
fixed-owned region · 49
forall · 26
\textit{FORALL} construct · 22
for all statement · 14
FORALL statement · 22
Fortran 77 · 1
Fortran 90 · 1
Fortran 90D · 14
Fortran D · 1
full search algorithm · 54
FUTURE project · 2
irregular assignment · 96
irregular owner · 33
iteration space · 22

K
Kali · 14; 83
kinds of templates · 121

L
language domain · 26
latency · 9
local iteration space · 33
locality of data · 8
local-to-local copy · 56
locks · 8
loop bound reduction · 61
loop variables · 22
loop-carried dependencies · 30
lowering · 30

M
majority computes rule · 66
mappable owner · 35
mapping · 26
mapping application · 34
mapping attribute · 25
mapping descriptor · 38
mapping extraction · 34
mapping model · 3; 159
mask absorption · 61; 74
mask piggybacking · 95
master processor · 46
master receives rule · 56
master sends rule · 56
master-sends-rule · 46
medium-level intermediate representation · 36
memory hole · 69
message · 9
MIMD · 2; 13
MIR · 36
MPI · 3; 10; 118; 159; 161
multi-owned scopes · 32
Multiple Instruction, Multiple Data · 13

\[ N \]

neighboring stencil assignment · 91
neighboring stencil reference · 91
network topology · 57
networks of workstations · 7
non-rectangular iteration spaces · 125
normalized volume assignment · 29
NOW · 7
np · 11
NUMBER_OF_PROCESSORS · 16
numerical intersection · 80

\[ O \]

ON directive · 21
ON HOME clause · 21
one-sided communication · 8
OpenMP · 8
owner · 20; 31; 33; 51; 55; 56
owner attribute · 25; 32
owner computes rule · 68
owner conflict · 41
owner generalization · 35; 50
owner mask · 61
owner privatization · 50
owner reduction · 50
owner selection list · 50
owner solution · 54
owner-aware · 37

\[ P \]

PAM · 10
Pandora · 14
parallel abstract machine · 4
Parallel Abstract Machine · 10
Parallel Engines Run-time System · 10; 45
parallel I/O · 15
parallelization-engine · 2
parent owner derivation · 50
PARTI libraries · 112
pattern cycle · 80
pattern cyclic enumeration · 84
PERTS · 10; 45
PGI-HPF · 115
pid · 11
polyhedron · 77
polyhedron scanning · 77
portability · 13
position equation · 70
post-SPMD optimizations · 87
PRE-HPF · 115
PREPARE · 1
pre-SPMD optimizations · 87
private variable · 99
procedure body · 32
process · 8
processor descriptor · 38
processor index vector · 38
processor subset · 108
PROCESSORS directive · 15
PROCESSORS_SHAPE · 16
PVM · 3; 10; 159; 161

\[ R \]

r, p, c-tuple · 18
rank · 15; 47
REALIGN directive · 19
receiving master · 56
receiving owner · 33
REDISTRIBUTE directive · 17
reduction · 92
regular dimension · 33
replicated · 9
replicated dimension · 27
row-wise compression · 73
run-time mapping descriptor · 28

\[ S \]

scalar assignment · 22
scatter assignment · 34; 96

175
schedule · 97
scope · 31
semaphores · 8
semi-regular assignment · 34
sending master · 56
sending owner · 33
shadow area · 92
shadow width · 92
shape · 16
shared bus · 9
shared memory · 7; 8; 9; 13
Single Program Multiple Data · 11
single-owning assignment · 33
single-owned scopes · 32
SMP · 7; 13
source level origin · 2
SPMD · 11
SPMD-engine · 37
squeeze function · 75
squeeze · 75
statement · 32
stencil assignment · 90
stencil offset · 90
SUPERB · 14
symbolic comparison · 65
symbolic intersection · 80
symbolic owner · 65
symmetric multiprocessing · 13
symmetric multiprocessors · 7
synchronization · 8; 10

TABLE directive · 16
thread · 8
tile-wise compression · 73
topology · 9
torus · 9
totally replicated · 54
transcriptive mapping · 28
two-owned assignment · 33; 34
two-owned scopes · 32
two-sided communication · 9; 10
two-sweep algorithm · 54

U
uniprocessors · 7

V
variable owner · 50
variable-owned regions · 49
Vienna Fortran · 1; 14
view · 35
volume-assignment · 22

W
WHERE statement · 22
wildcard replications · 57
work · 31
work directives · 25

table intersection · 80
task parallelism · 15