D I S S E R T A T I O N

Vienna Fortran 90 and its Compilation

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät
von

Siegfried Benkner
Wilhelminenstraße 19/11, 1160 Wien
Matr.Nr. 83 25 309
e-mail: sigi@par.univie.ac.at
geboren am 13. Juli 1965 in Baumgartenberg/Oberösterreich

Wien, im September 1994

Acknowledgements

I wish to thank my advisor, Prof. Hans Zima, for his support and guidance on my thesis and for the opportunity of working in his excellent research group at the University of Vienna.

Throughout the course of this research many people have contributed to this work. I am grateful to Peter Brezany for the many interesting discussions and significant contributions to this work. I also want to thank all the people from the PREPARE project who gave valuable feedback on my work. I would like to thank Michael Gerndt for introducing me into the internals of the SUPERB parallelization system.

I would particularly like to thank Peter Brezany, Piyush Mehrotra, Hans Moritsch, Bernd Wender and Ying Hou for partially proofreading my thesis and their helpful comments. Many thanks to all the people from the VFCS research group who contributed to this work and, in particular, to Bob Velkov for implementing a subset of Vienna Fortran 90 within the VFCS.

Finally, I would like to thank my parents, Viktoria and Erwin Benkner, for encouraging and supporting me throughout my years as a student in Vienna.

Abstract

In recent years massively parallel architectures, scalable to a large number of processors and offering spectacular peak performance ratios, have become increasingly important for solving large-scale scientific and engineering problems. However, because of the lack of a shared memory, these machines are difficult to program usually forcing the user to distribute the data across the processors and to insert message passing operations for communicating non-local data.

In this thesis, we define Vienna Fortran 90, an advanced data parallel language based on Fortran 90 and Vienna Fortran, that enables the user to program distributed memory architectures almost as easily as sequential computers using global addresses only. Vienna Fortran offers a variety of high level features for data distribution and thus combines the advantages of a shared memory programming paradigm with mechanisms for explicit user control of those aspects that have the greatest impact on efficiency. Vienna Fortran 90 provides a number of new features not found in any other comparable language such as HPF or Fortran D. This includes concepts for distributing user defined data structures, distribution of pointer objects, and a general framework for specifying arbitrary data and work distribution.

Having specified the language, we develop the major compilation techniques for restructuring Vienna Fortran 90 programs into parallel message passing programs for execution on distributed memory machines. These techniques are general enough to compile programs for a parameterized number of processors. From the user specified data distributions mathematical formulas are derived that determine the ownership and offsets of local data, the work distribution and the communication between processors. Novel techniques for the management of distributed data, including runtime representation of distributions, new methods for index conversion, and static as well as dynamic schemes for handling non-local data, have been developed.

In the early phases of parallelization high level communication descriptors are inserted into the code in order to describe potential data movements. Optimization of communication is performed on the level of communication descriptors. In the code generation phase communication descriptors are expanded into code to allocate buffers for non-local data, and to implement communication via message passing primitives.

To validate our techniques we have integrated a subset of Vienna Fortran 90 in the Vienna Fortran Compilation System and have implemented a prototype runtime support system.

Contents

List of Figures v

1 Introduction 1

1.1 Parallel Computers . . . 1

1.2 Programming Distributed Memory Parallel Computers . . . 2

1.2.1 Explicit Message Passing Languages . . . 3

1.2.2 Provision of Shared Memory . . . 4

1.2.3 Automatic Parallelization . . . 4

1.2.4 Language Extensions and Compilation Strategies . . . 6

1.3 Outline of the Thesis . . . 7

I Language 9

2 Vienna Fortran 90 11

2.1 Language Overview . . . 11

2.1.1 Fortran 90 Subset . . . 12

2.1.2 Vienna Fortran Language Extensions . . . 13

2.2 Language Model . . . 15

2.2.1 Basic Concepts . . . 15

2.2.2 Processors . . . 17

2.2.3 Data Distribution . . . 18

2.3 Processor Arrays . . . 21

2.3.1 Processor Array Declarations . . . 21

2.3.2 Processor Array References . . . 24

2.4 Data Distribution . . . 26

2.4.1 Direct Specification of Distributions . . . 26

2.4.2 Implicit Specification of Distributions . . . 37

2.4.3 Mapping Procedures . . . 44

2.4.4 Dynamic Data Distribution . . . 50

2.4.5 Allocatable Arrays . . . 54

2.4.6 Derived Types . . . 57

2.4.7 Pointers . . . 61

2.4.8 Distribution Type Definitions . . . 65

2.4.9 Defaults . . . 66

2.4.10 Undefined Distributions . . . 66
2.5 Procedures . . . 67
2.5.1 Interfaces . . . 67
2.5.2 Transfer of Distributed Arguments . . . 68
2.5.3 Distribution of Dummy Arrays . . . 69
2.5.4 Array Valued Functions . . . 74
2.6 Forall Loops . . . 74
2.7 Distribution Inquiries . . . 76
2.8 Modules . . . 77
2.9 Local Procedures . . . 77
2.10 Discussion . . . 78

II Compilation **81**

3 Model **83**
3.1 Basic Concepts . . . 83
3.2 Data Distribution . . . 84
3.3 Work Distribution . . . 85
3.4 Communication . . . 86
3.5 Index Conversion . . . 88
3.6 Composite Distributions . . . 88

4 Source to Target Mapping **90**
4.1 Introduction . . . 90
4.1.1 Source Language . . . 90
4.1.2 Target Language . . . 91
4.2 Basic Elements . . . 91
4.2.1 Computation of Local Index Sets . . . 92
4.2.2 Index Conversion and Memory Management . . . 101
4.2.3 Computation of Work Sets . . . 114
4.2.4 Computation of Communication Sets . . . 122
4.2.5 Handling Multidimensional Arrays . . . 134
4.2.6 Runtime Representation of Distributions . . . 135
4.3 Overview of the Compilation Process . . . 138
4.4 Basic Parallelization . . . 141
4.4.1 Introduction . . . 141
4.4.2 High Level Communication Descriptors . . . 141
4.4.3 Work Distribution . . . 144
4.4.4 Transformation of Executable Statements . . . 145

5 Program Analysis **162**
5.1 Flow Analysis . . . 162
5.1.1 Control Flow Analysis . . . 162
5.1.2 Data Flow Analysis . . . 163
5.2 Data Dependence Analysis . . . 163

5.3 Distribution Analysis 164
5.3.1 Distribution Analysis for Static Distributions 164
5.3.2 Distribution Analysis for Dynamic Distributions 165
5.4 Overlap Analysis 165
5.4.1 Computation of Overlap Descriptions 167
5.4.2 Overlap Concept for Direct Block-Cyclic Distributions 170
5.4.3 Overlap Concept for Implicit Block-Cyclic Distributions 175

6 Optimization and Code Generation **186**
6.1 Program Optimization 186
6.1.1 Communication Vectorization 187
6.1.2 Elimination of Communication Descriptors 188
6.1.3 Fusion of Communication Descriptors 191
6.1.4 Reuse of Communication Patterns 191
6.1.5 Latency Hiding 192
6.1.6 Overlap Communication 193
6.1.7 Storage Optimization 195
6.2 Structure of Target Code 195
6.2.1 Declaration Statements 195
6.2.2 Array Assignment Statements 197

7 Related Work **199**
7.1 Language Support 199
7.2 Parallelizing Compilers 202
7.3 Compilation Techniques 207

8 Conclusions **209**
8.1 Contributions 209
8.2 Future Research 212

Bibliography **214**

A Syntax **224**
A.1 Processor Array Declarations 225
A.2 Processor Array References 225
A.3 Type Declaration Statement 225
A.4 The DISTRIBUTED Attribute 225
A.5 The ALIGNED Attribute 227
A.6 Mapping Procedures 228
A.7 Dynamically Distributed Arrays 229
A.8 The DISTRIBUTE Statement 230
A.9 The ALIGN Statement 230
A.10 The ALLOCATE Statement 231
A.11 Derived Types 231
A.12 The LOCATION Attribute 232
A.13 Distribution Type Definitions 232

A.14 Forall . . . 233
A.15 The DCASE Construct . . . 233
A.16 The DISTRIBUTED, ALIGNED, DYNAMIC, CONNECT, LOCATION, RANGE, RESTORE Statements . . . 234

B Intrinsic Functions **235**

B.1 Intrinsic Distribution Function . . . 235
B.1.1 BLOCK(M) . . . 235
B.1.2 B_BLOCK(ARRAY) . . . 235
B.1.3 CYCLIC(M) . . . 235
B.1.4 INDIRECT(ARRAY) . . . 236
B.1.5 S_BLOCK(ARRAY) . . . 236
B.2 Distribution Inquiry Functions . . . 236
B.2.1 DISTRIBUTED(SOURCE) . . . 236
B.2.2 DYNAMIC(SOURCE) . . . 236
B.2.3 OWNED(A, PROCESSOR) . . . 237
B.2.4 OWNER(A, PROCESSOR) . . . 237
B.2.5 OWNERS(A) . . . 237
B.3 Processor Inquiry Functions . . . 238
B.3.1 $NP . . . 238
B.3.2 $MY_PROC . . . 238
B.4 Extended Fortran 90 Intrinsics . . . 238
B.4.1 LBOUND(ARRAY,DIM,PROCESSOR) . . . 238
B.4.2 UBOUND(ARRAY,DIM,PROCESSOR) . . . 239
B.4.3 SIZE(ARRAY,DIM,PROCESSOR) . . . 239

List of Figures

Distribution model . . . 20
Irregular processor array sections . . . 25
Block distributions . . . 29
General block distributions . . . 30
Block-cyclic distributions . . . 32
Replication over one processor array dimension . . . 34
Dimension permutation . . . 35
Indirect distribution . . . 36
Collapsed and replicated alignment . . . 39
Multilevel alignment . . . 43
Multiple recursive distribution . . . 47
Distribution of components of derived type . . . 58
Distribution of actual arrays . . . 68
Parallel sort and merge . . . 79
Representation of *CYCLIC(M)* distributions . . . 95
Alignment Trees . . . 97
Minimal part of local index set . . . 109
Memory allocation with gaps . . . 110
Index conversion for implicit block-cyclic distributions . . . 113
Work set of a regular array section . . . 119
Preprocessing overhead of communication set generation algorithms . . . 131
Compilation phases . . . 138
Semantics of communication descriptors . . . 143
Scalar communication descriptors . . . 144
Assumed shape and assumed distribution argument transfer . . . 155
Reshaping of array arguments at procedure boundaries . . . 157
Overlap communication for block-cyclic distributions . . . 171
Updating overlaps in shift computations . . . 175
Overlap communication for implicit block-cyclic distributions . . . 181
Overlap neighbors . . . 183
Overlap area of a 2-dimensional array . . . 185
Semantics of overlap communication descriptors . . . 193
Scheme for the implementation of array assignments . . . 198

Chapter 1

Introduction

1.1 Parallel Computers

Parallel computing is considered a key technology for solving the grand-challenge problems in science and engineering. In the last decade, a number of extremely powerful parallel supercomputer systems came on the market, offering a peak performance of currently up to several hundred GFlops and a memory capacity of several hundred Gbytes. These machines promise to provide solutions to many problems that require significantly more computing power than is available on conventional sequential processors. In particular, the provision of supercomputing power is crucial to tackle problems in application domains like large scale climate modeling, structure and galaxy formation, computational fluid dynamics, image processing, computer vision, reservoir modeling, molecular dynamics, and others.

From the architectural point of view, in particular with respect to the memory model, parallel computer systems can be classified into shared memory parallel computers (SMPC) and distributed memory parallel computers (DMPC).

Traditional shared memory supercomputers, like the Cray Y-MP, Cray C90 or the SGI Power Challenge Series, provide a global address space and each processor has uniform access to the shared memory. Whereas SMPCs offer a simple programming model by providing a global view of the memory, they cannot afford scalability due to the fact that only a relatively small number of processors can be connected efficiently to a shared memory. Therefore, these systems, also referred to as tightly coupled parallel systems, are unlikely to keep track with the ever increasing demand on processing power.

On the other hand, in a distributed memory system, such as the Intel Paragon, the Meiko CS-2, the nCube series, the Thinking Machines CM-5, or the Parsytec, each processor has its own local memory and usually may access data on other processors only via explicit message passing. Distributed memory parallel computers, also referred to as loosely coupled systems, are becoming increasingly important, since they are much easier to build, potentially scalable both in terms of memory and processing power, and offer the best cost/performance ratio of all supercomputers. In the near future, distributed memory systems consisting of thousands of processors will be capable of delivering teraflop performance.

Despite these advantages, distributed memory parallel computers are gaining user acceptance only very slowly, mainly because they generally do not provide tools for software development at a suitably high level. Distributed memory systems are difficult to program, forcing the user to deal with separate address spaces and low level details such as exchanging data between the processors. An additional reason for the current lack of popularity of DMPCs is that existing scientific and numerical applications are difficult to port to these machines.

Another class of supercomputer architectures attempts to combine the advantages of both SMPCs and DMPCs by providing virtual shared memory hardware support on top of a distributed memory system. In a virtual shared memory parallel computer, memory is globally addressable but physically distributed. However, unlike to a truly shared memory architecture access to memory is non-uniform; access to the local memory of a processor is faster than access to a remote memory. Examples of this class of architecture are the BBN TC2000 Butterfly, the Cray T3D, the Convex Exemplar, and the KSR-1.

The most important single difference between distributed memory systems and other computer architectures is the fact that the memory is physically distributed among the processors; the time required to access a non-local datum may be an order of magnitude higher than the time taken to access locally stored data. This has important consequences for program efficiency. In particular, the management of data, with the twin goals of both spreading the computational workload evenly and minimizing the delays caused when a processor has to wait for non-local data, becomes of paramount importance.

In order to exploit the tremendous computing power offered by today's distributed memory multiprocessors, support for the following aspects of software development has to be provided:

- Development of new programs at a higher level than message passing.
- Porting of existing sequential codes to parallel machines.
- Portability between various platforms.

1.2 Programming Distributed Memory Parallel Computers

Program development for distributed memory machines is much more complex than for shared memory machines. Programming such a machine efficiently requires a fairly detailed knowledge of the underlying architecture. In particular, exploiting the locality of data and computation is of paramount importance in order to achieve good performance. Most of our discussion will focus on the exploitation of *data parallelism*, which can be characterized by the simultaneous execution of the same operation on large data arrays, as opposed to control or task parallelism which is achieved through concurrent execution of different operations. In the following we take a closer look at different approaches to data parallel programming of distributed memory multicomputers. Basically these methods differ in the amount of information that has to be supplied by the user in order to enable the compiler to generate an efficient parallel program.

1.2.1 Explicit Message Passing Languages

Currently the dominant programming paradigm for scientific and numerical applications on such systems is based on conventional programming languages like Fortran and C with extensions for explicit message passing. With this approach the user has full control of the underlying hardware but is forced to cope with a number of low level details, while the task of the compiler is relatively easy.

The most widely used programming model for distributed memory machines is the *single program multiple data* (SPMD) or *data-parallel* model of computation in combination with the *owner computes paradigm*. With this model a parallel program is specified as a set of sequential processes that execute concurrently in a loosely synchronous fashion. Programs based on this model are characterized by a single thread of control; all processors execute the same program but on different parts of the data space. Programming with this model can be characterized as follows. First, the data arrays of the program are partitioned into pieces each of which is assigned to a processor which becomes the *owner* of the data assigned to it. Secondly, the work is distributed according to the specified data distribution by applying the owner computes paradigm, which implies that each processor only executes assignments to data elements owned by it. Finally, access to non-local data must be implemented by inserting appropriate communication statements to send and receive data between processors.

However, to write message passing programs along these lines that effectively exploit the full potential of a DMPC is not an easy task. This style of programming forces the user to explicitly manage all details of data distribution, work distribution and communication and to deal with a number of low level machine dependent details. The program has to be parameterized such that the data space is distributed among the local memories of the available processors. Furthermore, the user has to ensure that each processor only executes assignments to data it owns by distributing the computations appropriately. The interaction between the data allocated on different processors has to be explicitly specified by means of message passing commands and space to hold copies of non-local data has to be allocated. Each assignment to a distributed variable has to be transformed into a sequence of send and receive operations dependent on the chosen data distribution.

This results in a programming style which is tedious, time-consuming and error prone and which can be compared to assembly programming on a sequential machine. Usually message passing programs are up to ten times longer than their sequential counterparts, and may become extremely complex. Many new problems which are not present in sequential programming, like load balancing, synchronization, data incoherence, deadlock and livelock are encountered. Generally message passing programs are difficult to design, hard to debug, and inflexible, in the sense that it is not a trivial matter do change the underlying data distribution without having to modify large parts of the program. Because with this method the data distribution is hard-wired in all parts of the program, performance tuning and experimenting with different data distribution strategies usually requires a substantial rewrite of the program. Similar problems are encountered whenever such programs have to be adapted for a different distributed memory architecture. Hence, this style of programming has led to particularly slow software development cycles and, in consequence, high costs for software.

For all those reasons the state of the art in parallel software development is far behind the state of the art in parallel architectures. In the following we discuss several approaches that attempt to fill this gap and aim to overcome this 'parallel software crisis'.

1.2.2 Provision of Shared Memory

One approach to alleviate programming of distributed memory is characterized by the provision of a shared memory on top of a distributed memory machine by means of a software layer. This method allows to hide the details of the architecture from the programmer, and offers the user the easier programming model of a conventional shared memory parallel computer. The virtual shared memory is managed by the operating system by means of a page caching mechanism.

The virtual address space is organized in pages which are distributed over the local memories of the machine. Usually ownership of pages is distributed and may change dynamically. A database, either centralized or distributed, keeps track of the movement of pages. Access to a non-local datum generally results in the transfer of a whole page between processors. As a consequence, exploiting the locality of an algorithm is crucial for program performance. In order to minimize page movement between processors, sophisticated page caching techniques have to be applied, raising new problems like how to keep multiple copies residing on different processors coherent.

Whereas data movements are handled by the operating system, it is the task of the compiler to distribute the work (i.e. the loop iterations). The ultimate goal here is to utilize as much locality as possible and thus to minimize the needs for data transfers. Currently this task cannot be handled fully automatically and requires user assistance. Examples for this approach are language extensions such as SVM Fortran [16] or Fortran-S [25], which require the user to declare shared variables and parallel loops via compiler directives.

Currently a virtual shared memory realized solely by means of software introduces a large overhead. As a consequence, this approach may not be the right choice if maximum performance is the ultimate goal.

1.2.3 Automatic Parallelization

Another approach to overcome the 'parallel software crisis' is based on the development of supercompilers which automatically transform sequential programs or shared memory programs into programs that execute efficiently on distributed memory machines. Direct compilation of sequential programs for parallel machines or automatic parallelization shifts all of the complexity of extracting the parallelism inherent in an algorithm and generating an efficient parallel program from the user to the compiler.

Obviously this approach has several attractive advantages. It allows developing parallel programs almost in the same way as sequential programs. Users are not forced to learn how to write message passing programs but can continue using their familiar programming languages. The programmer can concentrate on the algorithmic issues of the program while the compiler handles all the low level details and performs machine dependent optimizations. Consequently, such programs would be easily maintainable and portable. Furthermore, the large number of existing sequential applications could be ported to parallel machines without the need of rewriting them.

Automatic parallelization can be seen as a further development of automatic vectorization [3, 83]. Vectorizing compilers take a sequential Fortran or C program and automatically transform it to execute efficiently on a conventional vector supercomputer. Without any doubt, the introduction of vectorizing compilers had a significant impact on the success of these machines.

There are, however, a number of difficulties with automatic parallelization and currently no such system is commercially available. Automatic parallelization is a much more complex task than automatic vectorization because a number of additional problems have to be solved. A major reason for the limited success of this approach is the fact that languages like Fortran or C, initially designed for conventional von Neumann architectures, do not provide constructs to express parallelism. As a consequence, the programmer is forced to impose a sequential execution order on its computation which is unnatural and obscures the parallelism inherent in many problems. Furthermore, many existing codes, especially those written in Fortran, are extremely difficult to analyze automatically since they often rely on sequence or storage association. For example equivalencing of arrays, a commonly used technique to reduce memory requirements, or the use of large work arrays to simulate dynamic memory allocation severely complicates the task of an automatic parallelizer. The situation is even worse when looking at C programs, mainly because of the unrestricted use of pointers, in particular address operations and pointer arithmetics.

In order for a compiler to detect the parallelism inherent in such programs, it must perform aggressive data-flow analysis, dependence analysis and interprocedural analysis. Furthermore, it requires a number of program transformations, in particular with respect to loops, in order to extract as much parallelism of a program as possible. Many of the problems encountered in automatic parallelization have been shown to be NP-complete and therefore heuristics or expert system support has to be provided [70]. Consequently, for the time being, only a small class of programs can be automatically parallelized with reasonably efficiency, and much work remains to be done.

So far, research in compiler technology and automatic parallelization has resulted in a number of prototype parallelization systems such as Kali [79, 93], Pandore [4], SUPERB [50, 51, 137], the MIMDizer [65], VFCS [34], and others have been developed. However, all these systems are semi-automatic or interactive parallelizers which heavily rely on user assistance. In particular they require the user to specify the distribution of the program's data among the processors of the parallel machine. This additional distribution information is then used by the compiler to restructure the program into a single program multiple data (SPMD) program by automatically determining the work distribution and generating the communication between the processors.

Exploiting the full potential of parallel computers requires a cooperative effort between the user and the compiler. In particular, selecting an appropriate data distribution is the most important step in automatic parallelization and significantly influences the efficiency of the generated parallel program.

Hence, a number of language extensions for conventional sequential programming languages like

Fortran and C have been developed which allow the user to explicitly specify the distribution of a program's data space among the processors of the parallel machine. This additional information alleviates the task of the compiler in generating an efficient distributed memory parallel program. In the following we examine this approach in more detail.

1.2.4 Language Extensions and Compilation Strategies

Over the last years there have been major efforts in developing programming languages that provide appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on sequential machines. Most of these approaches are based on sequential languages like Fortran or C with extensions for data distribution. This work includes Blaze [92], extensions to CM Fortran [114], CRAFT [103], Fortran D [48], the Yale Extensions [37], Kali [93], Vienna Fortran [139], and HPF [64].

All these languages offer the user a global address space, assume a single thread of control, and provide high level constructs that enable the user to specify the distribution of the data across the local memories of the target machine. Therefore, these languages are also referred to as data placement languages. Furthermore, some of these languages enable the user to indicate parallel computations and to control the distribution of loop iterations across the processors. A detailed discussion can be found in Section 7.1.

With this approach the user writes a sequential program and specifies at a high level how the data space of the program should be distributed. Usually the data distribution is specified by means of compiler directives or annotations added to the declaration of arrays, which constitute the central data structure in most scientific applications. Based on the user specified data distribution the compiler automatically derives an explicitly parallel SPMD program, i.e., it decides how to partition the control, sets up the necessary communication and generates the target code.

By analyzing the distribution annotations the compiler determines for each distributed array the set of local elements that have to be allocated on each particular processor. Based on this information it automatically partitions the control, usually by applying the owner computes rule, and translates global data references into local and non-local references. The non-local references are satisfied by inserting appropriate message-passing statements in the generated code. Since reduction of communication costs is crucial in achieving good performance, the communication is optimized where possible, in particular by combining communication statements, exploiting collective communication and by overlapping communication and computation in order to hide the communication latency.

This style of programming hides the machine specific details inside the compiler and enables the user to concentrate on the algorithmic issues. As a consequence, programs written in this way are not only much simpler than message passing programs but also more flexible. Changing the data layout requires only minor program changes. Furthermore, it eases the migration of existing sequential programs to distributed memory machines and the porting of codes between various different architectures.

1.3 Outline of the Thesis

In this thesis we define Vienna Fortran 90, an advanced data parallel language that provides a rich set of high level constructs for the specification of data distribution, and develop new techniques for compiling this language for efficient execution on distributed memory architectures.

One important goal of this thesis is to extend the language features of Vienna Fortran [139] to Fortran 90 and to develop advanced language features not found in any other of the above mentioned language extensions. In particular this includes concepts for distributing user defined data structures, distribution of pointers, methods for handling distributed data at procedure boundaries, and the provision of a general framework, based on mapping procedures, which can be used for both the specification of arbitrary data and work distribution. In addition to many other language extensions, Vienna Fortran 90 supports advanced numerical applications including adaptive multigrid codes, codes based on irregularly coupled meshes and unstructured mesh solvers. A detailed discussion of the features of Vienna Fortran 90 is presented in Chapter 2.

Moreover, we develop a general model for compiling Vienna Fortran 90 programs for efficient execution on distributed memory architectures, and formalize the major steps in the compilation process like data distribution, work distribution and communication generation. The parallelization is based on a source-to-source transformation from Vienna Fortran 90 to Fortran 90 plus calls to a message passing library. The user specified data distribution guides the compiler in restructuring the source program into a SPMD parallel message passing program.

The developed techniques are general enough to compile programs for a parameterized number of processors. In particular we derive mathematical functions, parameterized by the actual distribution of an array and the number of processors, to determine the ownership and offsets of local data, the work distribution and the necessary communication for arbitrary assignments to sections of distributed arrays. We define techniques for the management of distributed data, including runtime representation of distributions, index conversion functions, and static as well as dynamic schemes for the handling of non-local data.

In the early phases of parallelization high level communication descriptors are inserted into the code in order to describe potential data movements. Optimization of communication is performed on the level of communication descriptors. In the code generation phase communication descriptors are expanded into code to allocate buffers for non-local data, and to implement communication via message passing primitives.

To validate our techniques we have integrated a subset of Vienna Fortran 90 in the Vienna Fortran Compilation System and have implemented a prototype runtime support system.

The thesis is organized as follows:

This thesis is divided into two parts, one devoted to language design and the other to compilation techniques. In the first part we introduce Vienna Fortran 90. The language description is kept rather informal and most of the features are illustrated by means of examples.

In Section 2.2 we present a formal language model. Processor arrays are described in Section 2.3. In Section 2.4 we specify the various methods provided for the specification of data distribution, introduce the concept of mapping procedures, and extend the data distribution features to derived types and pointers.

Section 2.5 discusses procedures and describes the mechanisms provided by Vienna Fortran 90 in order to control the transfer of distributed arguments at procedure boundaries. In the remainder of the first part we discuss FORALL loops, distribution inquiries, modules and local procedures.

For the formal syntax of Vienna Fortran 90 the reader is referred to Appendix A. The Vienna Fortran 90 intrinsic functions can be found in Appendix B.

In the second part we describe the compilation of Vienna Fortran 90 for execution on distributed memory parallel computers.

In Chapter 3 we formulate a mathematical model which serves as the basis for the compilation process.

In Chapter 4 we describe the process of restructuring a Vienna Fortran 90 program into a parallel message passing program. We derive functions to compute data distribution, work distribution and communication. Furthermore, we derive functions used for transforming global indices into local indices, and discuss memory management with respect to distributed data. In particular we derive new techniques for the handling of implicit block-cyclic distributions considering arbitrary linear alignment functions. After that we deal with runtime representation of distributions and introduce layout descriptors. The basic parallelization phase, described next, introduces high level communication descriptors which provide an abstraction of the communication induced by the assignment to distributed variables.

Chapter 5 gives an overview of analysis techniques required for the compilation process and extends the overlap concept [54], which was initially developed for block distributions, to block-cyclic distributions.

In Chapter 6 we discuss various different optimization techniques that are essential for improving the efficiency of the parallel program.

A detailed discussion of related work is presented in Chapter 7. Finally, in Chapter 8, we summarize the contributions of this work, discuss the implementation status of Vienna Fortran 90 and conclude with a discussion of future research.

Part I

Language

Chapter 2

Vienna Fortran 90

2.1 Language Overview

Vienna Fortran 90 (VF90) is a machine independent, data parallel programming language that enables the user to write programs for distributed memory multiprocessors almost as easily as sequential programs. It is a subset of Fortran 90 with additional language constructs to support the formulation of data parallel algorithms. Most of these extensions are based on the language definition of Vienna Fortran [139] which served as one of the major inputs for the HPF language definition process [64].

Since performance of software on distributed memory machines is profoundly influenced by the manner in which data is distributed, Vienna Fortran 90 provides the user with a wide range of facilities for the mapping of data to processors. This includes direct specification of distributions, distribution by alignment or by referring to the distribution of another array, and the use of mapping arrays to support indirect and irregular distributions. Whereas the user writes a sequential program and specifies the distribution of the data, it is the task of the compiler to generate an explicitly parallel SPMD program by automatically determining the work distribution and the necessary communication.

Vienna Fortran 90 programs are written using global data references only. As a consequence, they are not only simpler than hand written message passing programs, but also considerably more flexible, enabling users to modify a data distribution without major reprogramming. Moreover, the features of Vienna Fortran also provide a powerful and convenient means for specifying parallel algorithms, which may exploit the properties of the chosen data distribution in order to obtain efficient code. Thus, Vienna Fortran combines the advantages of the shared memory programming paradigm with mechanisms for explicit user control of those aspects of the program which have the greatest impact on efficiency.

In the following we give a short overview of the main language features of Vienna Fortran 90. We assume that the reader is familiar with Fortran 90 [72] and thus mainly focus our discussion on the language extensions.

2.1.1 Fortran 90 Subset

Vienna Fortran 90 includes those language features of Fortran 90 which are most relevant in the context of data parallelism, i.e. the array sub-language. The use of array syntax does not only lead to a more intuitive programming style but can also significantly facilitate compilation and optimization. The *fetch-before-store* semantics of array assignment statements makes them to major sources of data parallelism. Other important features included are dynamic memory allocation, derived types, pointers and procedure interfaces. The Fortran 90 features included in Vienna Fortran 90 are listed in Table 2.1.

- Allocatable arrays
- Array sections
- Array assignments
- Array valued functions
- Array constants
- Array constructors
- Assumed shape arrays
- Automatic arrays
- Array Intrinsic Procedures
- Derived types
- Interfaces
- Internal procedures
- Intent specification
- Modules (restricted)
- Pointers
- WHERE statement and construct

Table 2.1: Summary of Fortran 90 features in VF90

Those elements of Fortran 90 which are not supported in the first implementation of Vienna Fortran 90 are listed in Table 2.2. All those marked with a '*' are redundant features of FORTRAN 77 in the sense that better methods exist in Fortran 90.

- Alternate return *
- BLOCK DATA program units *
- COMMON blocks *
- ENTRY statement *
- EQUIVALENCE statement *
- Statement functions *
- Structure constructors
- Module procedures
- Keyword arguments
- Optional arguments
- Overloading and generic features

Table 2.2: Fortran 90 features not included in VF90

2.1.2 Vienna Fortran Language Extensions

In this section we give a short overview of the language features that have been added to the above Fortran 90 subset. The major objective of these extensions is to give the user full control over the crucial task of selecting a distribution of the program's data over the processors of the parallel machine. The language extensions have been designed with a special focus on both alleviating the specification of data parallel algorithms and the compiler's complex task of program optimization. Most of these extensions are based on the Vienna Fortran language definition [139]. A comparison of Vienna Fortran 90 and Vienna Fortran is provided in Section 2.10.

The following language features are provided by Vienna Fortran 90:

- **Processor Arrays**

  Processor arrays are used to establish an abstraction from the actual underlying machine topology, and thus provide a means for achieving machine independence. The size of processor arrays may be specified either at compile time or at link- or runtime. In order to provide different views of the available set of processors, more than one processor array may be specified. Subsets of processor arrays may be referenced by means of Fortran 90 array sections. A one-dimensional default processor array is provided.

- **Data Distribution**

  The distribution of arrays with respect to the user defined processor structures can be specified using additional attributes within the type declaration statement. Distributions may be specified either directly, or implicitly by referring to the distribution of another array. For the most common regular distributions intrinsic functions are provided. Total as well as partial replication is supported. Irregular and indirect distributions may be specified by means of mapping arrays. Moreover, the user may specify arbitrary new distributions by means of *mapping procedures*.

- **Distribution of Derived Types**

  Components of objects of derived type may be distributed in much the same way as objects of intrinsic types. Furthermore, Vienna Fortran 90 provides special features which alleviate the mapping of hierarchically structured data objects to processor subsets.

- **Distribution of Pointers**

  Vienna Fortran 90 allows, however with some necessary restrictions, using pointers in conjunction with distributed arrays. This is especially important for data structures containing dynamically allocated arrays.

- **Procedure Interfaces**

  Interface blocks of external procedures may be extended with data distribution information and are extremely important for supporting separate compilation.

Dummy arrays may inherit the distribution from the actual arguments or may be distributed explicitly.

- **Modules**

  Modules have been extended in such a way that it is possible to define commonly used distributed data structures in a clean and portable way. These are called distribution modules.

- **Dynamic Data Distribution**

  Dynamic data distribution makes it possible to define and change the distribution of an array during runtime by explicit statements. Such arrays are called dynamically distributed arrays as opposed to statically distributed arrays which have a fixed distribution within their scope of declaration. Because of the significant impact of dynamic data distributions on the complexity of the compilation and optimization process there is a clear syntactic distinction between statically and dynamically distributed objects. A bundling mechanism in order to define sets of arrays that are always distributed together is provided.

- **Parallel Loops**

  Explicitly parallel loops, called *FORALL* loops, which do not contain any data dependences, may be specified. Private variables and special intrinsic reduction operators are provided. Moreover, it is possible to explicitly specify the distribution of the iterations of a forall loop to the available processors.

- **Inquiry Procedures**

  Vienna Fortran 90 provides several intrinsic procedures in order to obtain information about the actual distribution of a data object. This enables to user to write code finely tuned with respect to a certain data distribution.

- **Local Procedures**

  Local procedures which must not have any side-effects and which do not induce communication may be used. They provide a suitable means for reusing sequential kernels and interfacing with other sequential languages.

In the following we first specify some basic formal concepts and then describe the language extensions provided by Vienna Fortran 90 in more detail and discuss their impact on the chosen Fortran 90 subset. Syntax diagrams can be found in Appendix A and intrinsic functions in Appendix B.

2.2 Language Model

2.2.1 Basic Concepts

Definition 2.2.1: Allocation Instance, Allocation Status

- **Array Allocation Instance**
  An **allocation instance** *of an array is defined as the time interval in which the array is associated with storage. A* **statically allocated array** *has exactly one allocation instance, and its size must be declared at compile time. A* **dynamically allocated array** *may have more than one allocation instance, which corresponds to the time interval between the successful execution of an allocate statement and the subsequent deallocate statement.*
- **Allocation Status**
  The **allocation status** *of an array at a given time is* **allocated** *iff execution at that time is within an allocation instance for that array, otherwise it is* **deallocated** *or* **undefined**.

■

Fortran 90 provides the intrinsic function *ALLOCATED* to inquiry about an array's allocation status (see [72](14.8)). The function *ALLOCATED(ARRAY)* returns the value *.TRUE.* if the array is allocated, *.FALSE.* if it is deallocated. In case of an undefined allocation status the function result is also undefined.

Definition 2.2.2: Index Dimension

An index dimension is a non-empty, linearly ordered set of integers. It is called

- *a* **standard index dimension** *iff it can be represented in the form* $[l : u]$*, where* $l \leq u$ *and* $[l : u]$ *denotes the sequence of numbers* $(l, l+1, \ldots, u)$*,*
- *a* **normal index dimension**[1] *iff it is of the form* $[1 : u]$*,*
- *a* **regular index dimension** *iff it can be constructed from a standard index dimension through the application of a linear function of the form* $i{*}s{+}c$*,* $abs(s) > 1$*, to the elements of a standard index dimension and can be represented in the form* $[l : u : s]$*, where for* $l \leq u$ *and* $s > 0$*,* $[l : u : s]$ *denotes the sequence of numbers* $(l, l+s, \ldots, u)$*, and for* $l > u$ *and* $s < 0$ $[l : u : s]$ *denotes the sequence of numbers* $(l, l-s, \ldots, u)$*,*
- *an* **irregular index dimension** *in all other cases.*

■

[1] A normal index dimension is a also standard index dimension.

Definition 2.2.3: Index Domain

1. *Let D_i represent an index dimension. Then an* **index domain** *of rank n, denoted by* $\mathbf{I}$*, can be represented in the form* $\mathbf{I} = \mathbf{X}_{i=1}^{n} D_i$*, where* $\mathbf{X}$ *denotes the Cartesian product.* $\mathbf{I}$ *is called*
   - *a* **standard index domain** *iff all D_i are standard index dimensions,*
   - *a* **normal index domain** *iff all D_i are normal index dimensions,*
   - *a* **regular index domain** *iff at least one index dimension D_i is a regular index dimensions, and all others are standard index dimensions.*
   - *a* **irregular index domain** *iff at least one index dimension D_i, $1 \leq i \leq n$, is an irregular index dimension.*

   In the following, let $\mathbf{I}$ *denote an index domain of rank n, and i an integer number with $1 \leq i \leq n$.*
2. *The projection of* $\mathbf{I}$ *to its i-th component, D_i, is denoted by* $\mathbf{I}_i$*.*
3. $\mid D_i \mid$ *is the* **extent** *of dimension i (the number of elements along that dimension).*
4. *The* **shape** *of* $\mathbf{I}$ *is defined by* $shape(\mathbf{I}) := (\mid D_1 \mid, \ldots, \mid D_n \mid)$*.*

■

Definition 2.2.4: Index Mapping

An index mapping μ from an index domain $\mathbf{I}$ *to an index domain* $\mathbf{J}$ *is a total function that maps each element of* $\mathbf{I}$ *to a non-empty set of elements of* $\mathbf{J}$*.*

$$\mu : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{J}) - \{\phi\},$$

where $\mathcal{P}$ denotes the power set. ■

As will be seen later, index mappings are the basis for modeling data distributions.

Definition 2.2.5: Array Index Domains

Let in the following $\mathcal{A}$ denote the set of all data objects (scalars and arrays) declared in the program. Assume that $A \in \mathcal{A}$ is an arbitrary declared array.

1. *If the allocation status of A is* **allocated***, then:*
   (a) *A is associated with a standard index domain,* $\mathbf{I}^A$*. All attributes of* $\mathbf{I}^A$*, such as rank and shape, are applied to A with the same meaning as specified in Definition 3.*[2]
   (b) *$\mathcal{E}^A$ is the* **set of elements** *of A. The elements are scalar objects.*
   (c) *$index^A : \mathcal{E}^A \rightarrow \mathbf{I}^A$ is a function establishing a one-to-one correspondence between $\mathcal{E}^A$ and* $\mathbf{I}^A$*. For every array element $e \in \mathcal{E}^A$, $index^A(e)$ is called the* **index** *of e.*

[2] Whenever A is implied by the context, the superscript may be omitted. Analogous conventions hold for all similar cases.

2. *If the allocation status of A is* **deallocated**, *then only the rank of A is known. Neither an index domain nor a set of elements is associated with A.*

■

The index domain of an array is usually determined through its declaration. However, the declaration of an allocatable array only defines the array's rank. The shape and dimension bounds of dynamically allocated arrays are determined through the execution of an allocate statement and are only defined if the allocation status of the array is *allocated*.

Whereas in Fortran 77 only array elements or whole arrays may be referenced, Fortran 90 provides features for computing subarrays. A subarray can either be characterized as an array element or an array section. Let in the following A denote a whole array of rank n and assume that the allocation status of A is allocated[3]. A whole array always has a standard index domain associated with it.

Index Domain of an Array Section

Let us consider an array A of rank n with index domain $\mathbf{I}^A = [l_1 : u_1, \ldots, l_n : u_n]$. The index domain of an array section $A' = A(ss_1, \ldots, ss_n)$ is of rank n', $1 \leq n' \leq n$, iff n' subscripts ss_i are section subscripts (subscript triplet or vector subscript) and the remaining subscripts are scalars. Due to the Fortran 90 conventions the index domain of an array section can always be classified as a normal index domain. For example the index domain of the array section $A(11 : 100 : 2, 3)$ is given by $[1 : 45]$.)

The **extended index domain** of the array section A', denoted by $\bar{\mathbf{I}}^{A'}$, is given by $\bar{\mathbf{I}}^{A'} = [ss'_1, \ldots, ss'_n]$ where $ss'_i = ss_i$, if ss_i is a section subscript, and $ss'_i := [c_i : c_i]$ if ss_i is a scalar subscript that is denoted by c_i. The extended index domain of the array section A' can be classified as

- a standard index domain, if all *section subscripts* correspond to *subscript triplets* with stride 1
- a regular index domain, if all *section subscripts* correspond to *subscript triplets* with at least one subscript triplet having a stride $\neq 1$.
- an irregular index domain, if at least one *section subscript* is a *vector subscript*.

2.2.2 Processors

The set of processors on which a program will execute is denoted by $\mathbf{P}$. Vienna Fortran 90 uses the concept of processor arrays, in order to abstract from the actual machine topology of a distributed memory parallel computer, such as torus, mesh or hypercube. It is assumed that any two processors of $\mathbf{P}$ may communicate by exchanging messages. The mapping of a processor array to the physical machine is hidden in the compiler.

---

[3]If not stated otherwise, we assume in the following that the allocation status of an array is allocated.

2.2.3 Data Distribution

Vienna Fortran 90 requires the user to specify the distribution of the program's data to the set of processors. A distribution determines for each processor which data elements are in its local memory and which are not. Data distribution is modeled as a mapping of the elements of an array to (non-empty) sets of processors, including total as well as partial replication of data. Distributions may either be specified directly, by means of distribution functions, or implicitly by means of alignment functions.

2.2.3.1 Distribution Functions

A distribution function for an array maps each array element to one or more processors, which store those elements in their local memory. All elements mapped to a processor are said to be **owned** by that processor. Distribution functions are modeled as index mappings from an array index domain to a processor array index domain (see Definition 4).

A distribution function of an array A with respect to a processor array R is denoted by

$$\delta_R^A : \mathbf{I}^A \to \mathcal{P}(\mathbf{I}^R) - \{\phi\}.$$

A distribution function is called a **general** or **n-m distribution function** if it defines the distribution of an n-dimensional array with respect to an m-dimensional processor array. An important class of distributions are **composite distributions**. A composite distribution is specified by describing the distribution of each array dimension separately, without any interaction of dimensions, by means of **dimensional distribution functions** (i.e. $1-1$ distribution functions). A dimensional distribution function maps each index of an array dimension to one or more indices of a processor array dimension. For example, $\delta_{R_j}^{A_i} : \mathbf{I}_i^A \to \mathcal{P}(\mathbf{I}_j^R) - \{\phi\}$ is a dimensional distribution function of dimension i of A to dimension j of R.

2.2.3.2 Alignment Functions

Alignment functions are used to define the distribution of an array implicitly be establishing a relationship between its index domain and the index domain of another array. If for example A, B are different arrays used in the same computation (for example, within one assignment statement), then the relationship of their distributions may determine the amount of communication to be generated. To minimize communication elements of A and B used in the same computation should reside in the same processor memory. If δ^A and δ^B are in an invariant relationship with each other during a certain phase of program execution, then we say that A and B are aligned during that period. For example, if we know that $\delta^A = \delta^B$, then no communication needs to be generated for the array assignment $A = B + 1$. Alignment defines a mapping of the elements of an array B to a non-empty set of elements of an array A and therefore can be modeled in the same way as a distribution.

An alignment relationship of an array B with respect to an array A is established by means of an alignment function which is an index mapping from the index domain of array B to the index domain of array A. Such an alignment function is denoted by

$$\alpha_A^B : \mathbf{I}^B \to \mathcal{P}(\mathbf{I}^A) - \{\phi\}.$$

A is called **alignment source** array and B is called **alignee**[4].

As before we speak of a **composite alignment** if the alignment relationship is specified for each dimension separately (without any interaction of dimensions) by means of **dimensional alignment functions**.

The distribution of an alignee can be constructed by means of the distribution of the corresponding alignment source array as follows:

Definition 2.2.6: Construction of a Distribution

If $\mathbf{I}^A, \mathbf{I}^B, \delta_R^A$, *and* α_A^B *are given as above, then* δ_R^B *can be constructed as follows. For each* $\mathbf{i} \in \mathbf{I}^B$*:*

$$\delta_R^B(\mathbf{i}) := \bigcup_{\mathbf{j} \in \alpha_A^B(\mathbf{i})} \delta_R^A(\mathbf{j})$$

■

As a consequence, if index $\mathbf{i} \in \mathbf{I}^B$ is mapped via the alignment function α_A^B to an index $\mathbf{j} \in \mathbf{I}^A$, then it is guaranteed that the elements $B(\mathbf{i})$ and $A(\mathbf{j})$ are allocated on the same processor(s), regardless of the distribution of A.

2.2.3.3 Distribution Types

A **distribution type** specifies a *class of distributions* rather than individual distributions. It is determined by one distribution function or - in the case of composite distributions - a tuple of distribution functions together with the list of evaluated arguments. The application of a distribution type to an environment that specifies a data array and a processor array (section) yields a distribution. Therefore, a distribution is defined by three components: the index domains of a data array and a processor array (section) and distribution function. The distribution of an array A with respect to a processor array R can be described by means of a tuple $(I^A, I^R, type(\delta_R^A))$, where I^A and I^R are the index domains of an array and a processor array, respectively, and $type(\delta_R^A)$ denotes the distribution type.

Example 2.2.1: Distribution type

```
PROCESSORS R2(R,R)
REAL, DISTRIBUTED(BLOCK,CYCLIC) TO R2 :: A(N,N), B(M,M)
```

Given the above declarations the distribution type associated with (BLOCK,CYCLIC) is applied first to array A and processor array R2, and then to array B and R2. If N and M are different, the resulting distributions for A and B will be different. ■

[4] In [139] the term target array is used instead of alignee.

As shown in Example 2.2.1, (BLOCK,CYCLIC) represents a distribution type that can be applied to a two-dimensional data array and a two- or more-dimensional processor array (section) to yield a distribution.

If the distribution of an array is specified implicitly by means of alignment, the distribution type is determined by the associated distribution function, which can be obtained as shown in Definition 6, and the list of evaluated arguments including the alignment source array. This is discussed in more detail in Section 2.4.3.1.

Vienna Fortran 90 provides facilities to define distribution types in a similar way to derived data types (see Section 2.4.8) and intrinsic functions in order to inquire for an array's distribution type.

2.2.3.4 Static versus Dynamic Data Distribution

There is a clear separation between arrays that have a fixed distribution type within their scope, so-called **statically distributed arrays** and **dynamically distributed arrays** whose distribution may be modified within their scope. This strict syntactic separation between static and dynamic distributions supports the compiler in generating efficient target code. In Figure 2.1 the features for data distribution are illustrated.

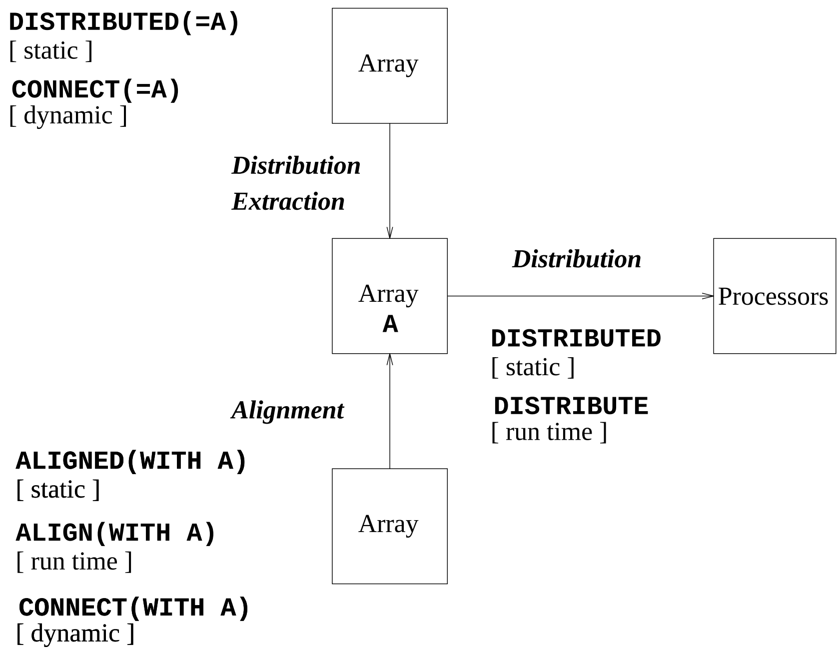


Figure 2.1: Distribution model

The distribution of statically distributed arrays is specified within their declaration by means of the *DISTRIBUTED* or *ALIGNED* attribute. By means of alignment or distribution extraction it is possible to specify the distribution of an array implicitly by referring to the distribution of another array. Depending on their allocation scheme statically distributed arrays they can be divided in two classes. Statically distributed and statically allocated arrays always have a fixed distribution within their scope of declaration. On the other hand, a statically distributed and dynamically allocated array always has a fixed distribution type but its distribution may change due to modifications of its index domain caused by the execution of an allocate statement. An alignment relationship between two statically distributed arrays is valid throughout the scope of the declaration (see Section 2.4.2.1 and 2.4.2.2) as long as none of them is reallocated.

Dynamically distributed arrays must be declared with the *DYNAMIC* attribute and are usually associated with a distribution by execution of a *DISTRIBUTE* statement, an *ALIGN* statement or as the result of a procedure call. The *CONNECT attribute* is provided to specify a run-time invariant relationship between the distributions of groups of dynamically distributed arrays. In Figure 2.1 the data distribution model of Vienna Fortran 90 is depicted.

2.3 Processor Arrays

Processor arrays provide a suitable means for defining and accessing the underlying set of processors using the same mechanisms as for 'ordinary' data arrays. By declaring several processor arrays, it is possible to impose one or more rectilinear structures on the underlying set of processors **P**, and thus providing different views of the set of processors. Correspondence between different processor arrays is established by means of the Fortran element order. Arbitrary subsets of **P** may be accessed and referenced using array subscripts or array sections.

2.3.1 Processor Array Declarations

The user may declare one or more processor arrays by means of the *PROCESSORS* statement. Syntactically, the declaration of a processor array is very similar to a Fortran 90 array declaration. Processor array declarations may appear in the specification part of a program unit, only. The processor arrays of the main program unit define the set of processors **P** on which the program will execute. If in a program unit more than one processor array is declared, the textually first one is called the **primary processor array**, all others are **secondary processor arrays** or reshapes of the primary processor array.

The total number of processors on which the program executes may be accessed by the Vienna Fortran intrinsic function *$NP*. Vienna Fortran always provides a one-dimensional *implicit processor array $P* of the form *$P(1:$NP)*. *$P* is accessible in all program units and procedures. If a program unit does not contain a processor declaration statement, *$P* is the primary processor array and no secondary processor arrays exist.

All processor arrays refer to the same set of processors and thus the size of any two explicitly declared processor arrays in a program unit must be the same. As a consequence, the size of processor arrays occurring in internal procedures must match the size of the processor array of the host program unit. For external procedures this restriction is relaxed; the size of a processor array of an external procedure must be equal or less *$NP*.

Example 2.3.1: Processor array declarations

```
PARAMETER  (NP1 = 10)        ! number of processors fixed at compile time
PROCESSORS, DIMENSION(NP1,NP1) :: P2
```

In this example the number of processors on which the program will execute is declared to be 100 and the individual processors may be accessed by the two-dimensional processor array P2 using ordinary array subscripts.

```
PROCESSORS :: PP($NP/4, 4) ! number of processors unknown at compile time
```

The intrinsic function $NP is used in the processor array bounds expressions, and therefore the program will execute on any number of processors that is divisible by 4. If $NP is not divisible by 4, then the size of PP will not be equal to the size of the implicit processor array $P and a runtime error message will be issued.

```
PROCESSORS, ASSERT(N >= 8) :: R2(N,N), R1(0:N*N-1)
```

Here a two-dimensional processor array R2 and a one-dimensional processor array R1 with N^2 processors is declared. The value of N is asserted to be greater than or equal 8, so the minimum number of processors of the physical machine must be 64. N is a symbolic constant, whose value is determined at load time depending on the number of processors actually available. The individual processors may be accessed in the same way as ordinary array elements by use of subscripts. R1 provides a one-dimensional view of the set of available processors. For all i, j with $1 \leq i, j \leq N, R2(i, j)$ is mapped to $R1(i - 1 + (j - 1) * N)$, so processor R2(1,1) corresponds to processor R1(0). R2 is the primary processor array and R1 the secondary processor array. ■

For a program that should execute on a particular number of processors, which is known at compile time, the bounds of the processor declaration(s) must be compile time constants. If the number of processors is known at compile time, the compiler may produce more efficient code.

A parameterization of the program by the number of processors can be achieved in the following two ways:

- The intrinsic function *$NP* may be referenced in the dimension bound expressions.
- Scalar variables which do not fall into the category of specification expressions [72](7.1.6.2) may be used in the dimension bound expressions. The values of such variables are obtained from the environment upon program start (e.g. through the use of command-line arguments).

A local processor array of an external procedure is called an *automatic processor array* if the bound expressions contain references to dummy arguments. Such a processor array may have a different size in different incarnations of the procedure. However, the size of an automatic processor array must always be less than or equal *$NP*.

Assertions may play an important role in processor declarations by supplying the compiler with additional information on the range of available processors and the values of variables occurring in dimension bound expressions. This may enable the compiler to avoid worst case assumptions or runtime checks and thus more efficient code may be produced. Assertion may be specified by means of the *ASSERT* attribute followed by a logical expression as shown in Example 2.3.1.

2.3.1.1 Scope of Processor Array Declarations

The scope of a processor declaration is in accordance with the Fortran 90 Standard scoping rules [72](14.1.2). Processor arrays in this sense are local entities of class (1) and therefore must not have the same name as other entities of class (1) of the same scoping unit. (e.g named variables, named constants, internal procedures, dummy procedures, derived types, etc.)

In internal procedures the processor arrays of the host scoping unit are accessible by *host association* [72](12.1.2.2.1). Processor declarations in internal procedures can be used to declare local reshapes of the processor arrays of the host program unit or to redefine the primary processor array as shown in Example 2.3.2.

Example 2.3.2: Procedure local processor arrays

```
PROGRAM MAIN
...
PROCESSORS R2(P,P), R1(P*P)
  ! R2 is primary processor array;  R1 is secondary processor array
...
CONTAINS
   SUBROUTINE SUB( ... )
      PROCESSORS RL(0:P-1,0:P-1)
         !  RL is primary processor array
         !  R2, R1 are secondary processor arrays (host association)
   ...
```

In the main program unit, R2 is the primary processor array and R1 is the secondary processor array. Inside the internal subroutine SUB all three processor arrays RL, R2, and R1 are accessible, however RL is now the primary processor array while R2 and R1 are secondary processor arrays. ∎

Processor arrays declared in modules may be made accessible by means the Fortran 90 mechanism of *use association* [72](12.1.2.2.2). Modules provide a clean method for sharing processor array declarations between different program units. Note that Vienna Fortran 90 does not allow to explicitly pass processor arrays as arguments to procedures. However, the full set of processors is accessible in each program unit through the use of the implicit default processor array *$P*.

2.3.2 Processor Array References

Processor array references are used in contexts which require that either individual processors or sequences of processors are explicitly specified. This includes the *DISTRIBUTED* attribute (Section 2.4.1.1), the *DISTRIBUTE* statement (Section 2.4.4.4), and the work distribution specification in *FORALL*-loops (Section 2.6).

A processor reference designates either a whole processor array, an element of a processor array, or a section of a processor array. Processor subsets can be specified by using arbitrary array sections of Fortran 90. For example rows, columns, rectilinear blocks and regular discontiguous regions of a two-dimensional processor array can be described by means of subscript triplets. In addition, irregular subsets of a processor array can be described by means of vector subscripts.

Example 2.3.3: Regular processor array sections

```
PARAMETER (N=10)
PROCESSORS R2(N,N),R1(N*N)

R1(::2)                      ! all processors with odd number
R1(2::2)                     ! all processors with even number
R2(I,:)                      ! i-th row of R2
R1(N:1:-1)                   ! R1 in reverse order
R2(1:N/2,1:N/2)              ! upper left quarter of R2
R2(/2,1/)                    ! R2 transposed
RESHAPE(R,(/25,4/))          ! 25x4 reshape of R2
R2(3:1:-2,6:1:-3)(/2,1/)     ! R2(3,6), R2(3,3), R2(1,6), R2(1,3) in this order
```

■

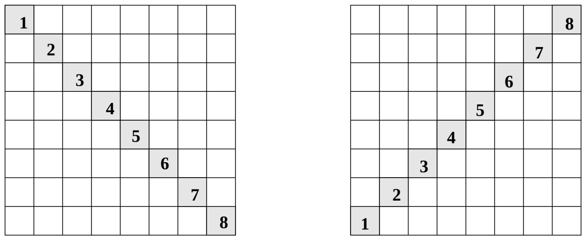
The dimensions of a processor array (section) may be permuted by supplying a so-called *dimension permutation* after a processor reference. If a dimension permutation is present, its size must match the rank of the corresponding processor array section. Given a processor reference of the form $R(/d_1, ..., d_m/)$ where R is a processor array (section) of rank m, then the dimension permutation $(/d_1, ..., d_m/)$ specifies a permutation of $(1, ..., m)$ and the evaluation of this construct yields a processor array R' of rank m such that for all $j, 1 \leq j \leq m$, dimension j of R' corresponds to dimension d_j of R.

Additional reshapes of a processor array may be obtained by means of the Fortran 90 intrinsic function *RESHAPE* [5].

Irregular processor array sections may be formed by using Fortran 90 vector subscripts, which allow to specify any arbitrary subset of the processor set. Using vector subscripts for example, the diagonals of a two-dimensional processor array can be selected by referring to a one-dimensional processor array, as shown in the Example 2.3.4.

[5] In this case, the optional arguments of *RESHAPE* (i.e *PAD* and *ORDER*) must not be present.

Example 2.3.4: Irregular processor array sections



(a) (b)

```
PARAMETER (N=8)
PROCESSORS R2(N,N), R1(N*N)
INTEGER,PARAMETER,DIMENSION(N) :: DIAG_1 = (/ ( (I-1)*N+I, I=1,N) /), &
                                  DIAG_2 = (/ ( I*(N-1)+1, I=1,N) /)

... R1(DIAG_1)   ! designates 1. diagonal of R2  (Fig. a)
... R1(DIAG_2)   ! designates 2. diagonal of R2  (Fig. b)
```

This example shows how the diagonals of the two dimensional processor array R2 can be referenced via a vector subscripted array section of processor array R1. ∎

2.3.2.1 Processor Array Inquiry Functions

The Fortran 90 intrinsic functions $SIZE$, $LBOUND$, $UBOUND$, and $SHAPE$ may be used in Vienna Fortran 90 with a processor array as argument. The index of an executing processor in *\$P* can be inquired by the Vienna Fortran intrinsic function *\$MY_PROC*.

2.3.2.2 Restrictions on the use of Processor Arrays

Processor arrays may be referenced only within the declaration of distributed objects, in the *DISTRIBUTE* statement, in the work-distribution specification of *FORALL* loops, and as arguments of the processor array inquiry functions listed above.

Moreover, processor arrays must not be passed as arguments to procedures. A globally accessible processor arrays has to be declared within a module which must be *use-associated* by all program units which need to access this processor array.

2.4 Data Distribution

Vienna Fortran 90 requires the user to specify the distribution of the program's data to the set of processors. A data distribution determines for each processor which data elements are in its local memory and which are not. The distribution of data objects for which a distribution has not been specified by the programmer is automatically determined by the compiler[6].

The data distribution is used by the compiler to guide the process of restructuring the code into an SPMD parallel program. Since the distribution of data is crucial for the performance of the resulting parallel program, Vienna Fortran provides a variety of methods to distribute data. This includes direct, indirect and implicit specification of distributions.

Vienna Fortran defines a distribution as a mapping of the elements of an array to (non-empty) sets of processors, including total as well as partial replication of data. Only objects of data type other than *CHARACTER* may be distributed.

2.4.1 Direct Specification of Distributions

A direct distribution of an array specifies a **one level mapping** of the array elements to non-empty sets of processors.

2.4.1.1 The DISTRIBUTED Attribute

The *DISTRIBUTED* attribute is used to specify the distribution of an array within the array's declaration. It may be specified either within the type declaration statement or by means of the *DISTRIBUTED statement* in the specification part of a program unit (see Example 2.4.1).

In case of a statically distributed array it specifies a distribution type that is invariant within the array's scope of declaration. For dynamically distributed arrays, which must also have the *DYNAMIC* attribute, it may be used in order to provide an initial distribution (see Section 2.4.4.).

The *DISTRIBUTED* attribute consists of the keyword *DISTRIBUTED* followed by a *distribution specification*[7]. A distribution specification consists of a *distribution type specification* and an optional processor reference which designates after the keyword *TO* the processor array (section) to which the distribution refers. If no processor reference is present, the primary processor array is chosen by default[8].

A *composite distribution type specification* for an n-dimensional array usually consists of a list of *n dimensional distribution type specifications*. Each dimensional distribution type specification is either a distribution function reference which determines how the corresponding array dimension should be distributed or a ':' which indicates that the corresponding array dimension is not distributed and thus replicated. A *general distribution type specification* for an n-dimensional array with

[6] The compiler for example may choose to replicate (i.e. allocate on each processor) such objects.
[7] In [139] this is referred to as *distribution expression*.
[8] This is not the case for dummy arrays with an assumed distribution.

respect to an m-dimensional processor array must be a reference to a $n - m$ distribution function.

For the most common distributions intrinsic functions are provided. The user may define arbitrary new distributions by means of so-called mapping procedures which are discussed in Section 2.4.3.

Example 2.4.1: The *DISTRIBUTED* attribute

```
PARAMETER (N=2)
PROCESSORS :: R1(N*N), R2(N,N)

REAL, DISTRIBUTED(BLOCK,:)              :: A(100,100)
REAL, DISTRIBUTED(MYFUNC)  TO R2        :: B(100,100,100)
REAL, DIMENSION(100)                    :: C

DISTRIBUTED(BLOCK) :: C
```

In this code fragment two processor arrays are declared. $R1$ is the primary processor array and $R2$ is the secondary processor array. The first dimension of A is partitioned into 4 blocks of length 25. whereas the second dimension stays unpartitioned. The resulting blocks of rows are mapped by default to the primary processor array $R1$. More exactly, the array section $A((I-1)*25+1 : I*25, :)$ is mapped to processor $R(I)$, $1 \leq I \leq 4$. Array B is distributed by means of a user-defined distribution function to the processor array $R2$. Note that $MYFUNC$ must designate a $3-2$ distribution function. The distribution of C is specified by means of the DISTRIBUTED statement.
∎

The *DISTRIBUTED* attribute is evaluated at the time of declaration to yield a distribution type. All expressions occurring as arguments of distribution functions must be specification expressions.

In the following we first define the intrinsic distribution functions provided by Vienna Fortran 90 in the context of one-dimensional arrays and then define the semantics of composite distributions.

Intrinsic Dimensional Distribution Functions

The intrinsic distribution functions discussed in the following are $1-1$ distribution functions since they specify the mapping of one array dimension to one processor array dimension. Let N denote the size of an array dimension and P the size of the corresponding processors array dimension.[9]

1. Basic Block Distributions

Block distributions are specified by the intrinsic function *BLOCK[(M)]*. The semantics of this distribution function is dependent on whether the block size M is explicitly specified or not.

[9] Without loss of generality we assume for sake of simplicity that the corresponding index dimensions of the array and the processor array have been normalized to $[1:N]$ and $[1:P]$, respectively.

(a) BLOCK

If the optional block size M is not specified, then the array is divided into contiguous blocks, whose sizes differ by at most 1.

More precisely, let $r := (N \ mod \ P)$, and $q := \left\lfloor \frac{N}{P} \right\rfloor$. The array is divided into r blocks of size $q+1$ — which are mapped to the first r processors — and $P-r$ blocks of size q, mapped to the remaining processors. Therefore we have

$$N = q.P + r = (q+1)r + q(P-r), \text{ where } 0 \leq r < P.$$

If $r = 0$, then the array is divided into P blocks of size q. The distribution function δ thus can be defined as follows:

Let $q := \left\lfloor \frac{N}{P} \right\rfloor$, $r := (N \ mod \ P)$, and $1 \leq i \leq N$:

$$\delta(i) = \begin{cases} \left\{ \left\lceil \frac{i}{q+1} \right\rceil \right\} & \text{iff } 1 \leq i \leq (q+1)r \\ \\ \left\{ \left\lceil \frac{i-r}{q} \right\rceil \right\} & otherwise \end{cases} \tag{2.1}$$

In the case were $(N \ mod \ P) = 0$ this can be simplified to:

$$\delta(i) = \left\{ \left\lceil \frac{i}{q} \right\rceil \mid 1 \leq i \leq N \right\} \tag{2.2}$$

(b) BLOCK(M)

If the block size M is specified explicitly, the restriction $M \geq \left\lceil \frac{N}{P} \right\rceil$ must always be true. This guarantees that the array is partitioned into at most P blocks, and therefore no wrap around as in the case of cyclic distributions (see below) will occur. The array is divided into $n = \left\lfloor \frac{N}{M} \right\rfloor$ contiguous blocks of size M which are mapped to the first n processors, with $0 \leq n \leq P$. If N is not exactly divisible by M, then the remaining $(N \ mod \ M)$ elements are mapped to processor $n+1$. All other processors (if any) do not receive any elements. Therefore we have

$$N = M.n + (N \ mod \ M)$$

The distribution function δ for a *BLOCK(M)* distribution thus is defined as follows:

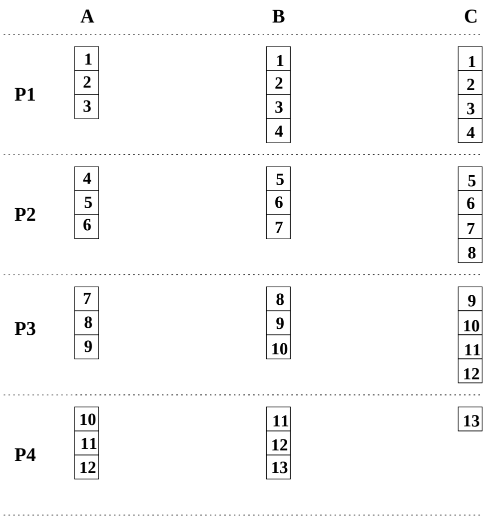
$$\delta(i) = \left\{ \left\lceil \frac{i}{M} \right\rceil \mid 1 \leq i \leq N \quad \wedge \quad M \geq \left\lceil \frac{N}{P} \right\rceil \right\} \tag{2.3}$$

Whereas the distribution function associated with a *BLOCK* distribution is more complex than that of a $BLOCK(M)$ distribution, it might result in a better load balancing. Block distributions are illustrated in Example 2.4.2.

Example 2.4.2: Block distributions

```
PROCESSORS :: R1(4)

DISTRIBUTED(BLOCK)    :: A(12)
DISTRIBUTED(BLOCK)    :: B(13)
DISTRIBUTED(BLOCK(4)) :: C(13)
```



For array A, we have $q = 3$ and $r = 0$. For B, we have $q = 3$ and $r = 1$ For the third array C a block size of 4 is explicitly specified; we have $n = 3$ blocks of size 4 and one block of size 1. This results in the following distribution functions:

- $\delta^A(i) = \{1\}$, $1 \leq i \leq 3$
- $\delta^A(i) = \{2\}$, $4 \leq i \leq 6$
- $\delta^A(i) = \{3\}$, $7 \leq i \leq 9$
- $\delta^A(i) = \{4\}$, $10 \leq i \leq 12$
- $\delta^B(i) = \{1\}$, $1 \leq i \leq 4$
- $\delta^B(i) = \{2\}$, $5 \leq i \leq 7$
- $\delta^B(i) = \{3\}$, $8 \leq i \leq 10$
- $\delta^B(i) = \{4\}$, $11 \leq i \leq 13$.
- $\delta^C(i) = \{1\}$, $1 \leq i \leq 4$
- $\delta^C(i) = \{2\}$, $5 \leq i \leq 8$
- $\delta^C(i) = \{3\}$, $9 \leq i \leq 12$
- $\delta^C(i) = \{4\}$, $i = 13$.

■

2. General BLOCK Distributions

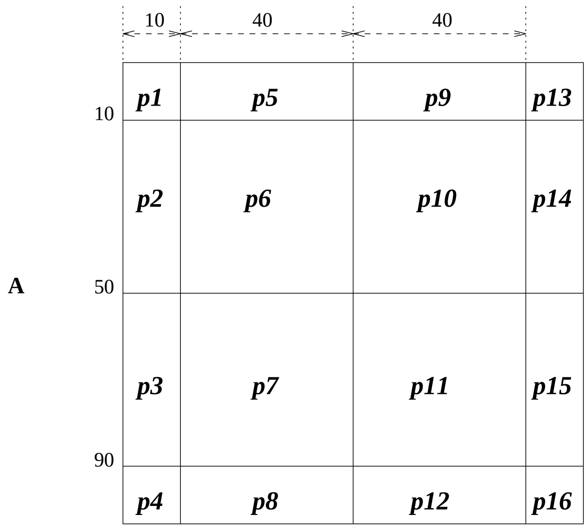
For the specification of more general block distributions the intrinsic functions B_BLOCK and S_BLOCK which both take a one-dimensional integer array as argument are provided. Both functions are used to specify the mapping of one array dimension to one processor array dimension.

In the following let $A(LB : UB)$ denote a one dimensional array, that should be distributed to a processor array with P processors of the form $R(1 : P)$. Let furthermore $G(1 : P - 1)$ denote a one-dimensional integer array.

(a) B_BLOCK

The distribution function $B\_BLOCK(G)$ partitions the array A into P contiguous blocks by interpreting each element $G(j)$, $1 \leq j < P$ as upper bound of block j. The index ranges associated with the blocks are given as follows:

Example 2.4.3: General block distributions




```
PROCESSORS R(4,4)

INTEGER, DIMENSION(3), PARAMETER :: BB = (/ 10,50,90 /)
INTEGER, DIMENSION(3), PARAMETER :: SB = (/ 10,40,40 /)

REAL, DISTRIBUTED(B_BLOCK(BB),S_BLOCK(SB)) :: A(100,100)
```

The above declaration of array A results in the depicted distribution pattern. Note that in this example the distribution functions B_BLOCK(BB) and S_BLOCK(SB) determine equivalent distributions of both dimensions. ■

Block 1: $[LB : G(1)]$
Block j, $1 < j < P$: $[G(j-1)+1 : G(j)]$
Block P: $[G(P-1)+1 : UB]$

The values of array G must be defined in such a way that each block is associated with a nonempty index range. The resulting distribution function is given by:

$$\delta(i) = \begin{cases} 1, & \text{if } LB \leq i \leq G(1) \\ j, 1 < j < P, & \text{if } G(j-1)+1 \leq i \leq G(j) \\ P, & \text{if } G(P-1)+1 \leq i \leq UB \end{cases} \tag{2.4}$$

(b) S_BLOCK

The distribution function $S\_BLOCK(G)$ partitions the array dimension into P contiguous blocks. For all $i, 1 \leq i < P$, $G(i)$ specifies the size of block i. The index ranges associated with the blocks are given as follows:

Block 1: $[LB : LB + G(1) - 1]$
Block 2: $[LB + G(1) : LB + G(1) + G(2) - 1]$
...
Block P: $[\ldots : UB]$

The values of G must be specified in such a way that all index ranges are nonempty.

3. Block-Cyclic and Cyclic Distributions

Block-cyclic distributions are specified by the intrinsic distribution function *CYCLIC(M)*, with an optional argument, $M \geq 1$, of type integer. *CYCLIC(M)* partitions an array dimension into contiguous segments of length M and maps them in a round-robin fashion to the processors. If the block length M is not specified, its default value is 1. In this case, we speak of a cyclic distribution, rather than of a block-cyclic distributions.

The distribution function defined by the block-cyclic distribution function *CYCLIC(M)* is given as follows:

$$\delta(i) = \left\{1 + \left\lfloor \frac{i-1}{M} \right\rfloor \; mod \; P \;|\; 1 \leq i \leq N \right\} \tag{2.5}$$

which in case of a *CYCLIC* distribution, where $M = 1$, can be simplified to:

$$\delta(i) = \{1 + (i-1) \; mod \; P \;|\; 1 \leq i \leq N\} \tag{2.6}$$

Note that for $M \geq \left\lceil \frac{N}{P} \right\rceil$ a *CYCLIC(M)* distribution is equivalent to a *BLOCK(M)* distribution. Block-cyclic distributions are illustrated in Example 2.4.4.

Example 2.4.4: Block-Cyclic distributions

```
PROCESSORS :: R1(4)

DISTRIBUTED(CYCLIC)    :: A(13)
DISTRIBUTED(CYCLIC(2)) :: B(13)
DISTRIBUTED(CYCLIC(4)) :: C(13)
```

	A	B	C
P1	1, 5, 9, 13	1, 2, 9, 10	1, 2, 3, 4
P2	2, 6, 10	3, 4, 11, 12	5, 6, 7, 8
P3	3, 7, 11	5, 6, 13	9, 10, 11, 12
P4	4, 8, 12	7, 8	13

The resulting distribution functions are as follows:

- $\delta^A(i) = \{1\}$, $i = 1, 5, 9, 13$
- $\delta^A(i) = \{2\}$, $i = 2, 6, 10$
- $\delta^A(i) = \{3\}$, $i = 3, 7, 11$
- $\delta^A(i) = \{4\}$, $i = 4, 8, 12$
- $\delta^B(i) = \{1\}$, $i = 1, 2, 9, 10$
- $\delta^B(i) = \{2\}$, $i = 3, 4, 11, 12$
- $\delta^B(i) = \{3\}$, $i = 5, 6, 13$
- $\delta^B(i) = \{4\}$, $i = 7, 8$
- $\delta^C(i) = \{1\}$, $1 \leq i \leq 4$
- $\delta^C(i) = \{2\}$, $5 \leq i \leq 8$
- $\delta^C(i) = \{3\}$, $9 \leq i \leq 12$
- $\delta^C(i) = \{4\}$, $i = 13$.

■

Distribution of Multidimensional Arrays

If the distribution of a multidimensional array is specified in such a way that each array dimension is distributed separately, then we speak of a **composite distribution**. Such a distribution can be represented as a tuple of independent mappings between single array and single processor array dimensions. Composite distributions are specified by a list of *dimensional distribution type specifications*, one for each array dimension. A dimensional distribution type specification can be a reference to a dimensional distribution function, a distribution extraction, or the elision symbol ":" which indicates that the dimension will not be distributed.

The number of distributed dimensions[10] of an array is called its **distribution rank** and must not be greater than the rank of the processor array to which the array is mapped. Replication of distributed data takes place in those cases where an array's distribution rank is smaller than the rank of the corresponding processor array (section). In the following we precisely define the semantics of a composite distribution.

[10] These are those dimensions for which a distribution type specification other than ":" is provided.

Definition 2.4.1: Composite Distributions

Let us consider the following form of the DISTRIBUTED *attribute*

$\mathtt{DISTRIBUTED}(dts_1, \ldots, dts_n)$ TO $\mathtt{R}(ss_1, \ldots, ss_m)(/\pi_1, \ldots, \pi_{m'}/)$ :: $\mathtt{A}(N_1, \ldots, N_n)$

which specifies the distribution of an n-dimensional array A with associated index domain $\mathbf{I}^A = [1 : N_1, \ldots, 1 : N_n]$ *to a section* R' *of a m-dimensional processor array R with index domain* $\mathbf{I}^R = [1 : R_1, \ldots, 1 : R_m]$.

1. *Each* dts_i, $1 \leq i \leq n$, *is either*
   - *a reference to a distribution function*
   - *or the symbol ':'.*

   The distribution rank n', $0 \leq n' \leq n$, *is given by the number of* dts_i *that are references to distribution functions. Let* $dim_\delta = (d_1, \ldots, d_{n'})$ *denote the sequence of distributed dimensions.*
2. *Each* ss_j, $1 \leq j \leq m$, *is either*
   - *a scalar subscript or*
   - *a subscript triplet or*
   - *a vector subscript*

   Let $R' = R(ss_1, \ldots, ss_m)$ *denote the processor array section and* $\bar{\mathbf{I}}^{R'}$ *its extended index domain. The number of* ss_j *that are section subscripts determine the rank* m' *of the processor array section. Let* $(d'_1, \ldots, d'_{m'})$ *denote the sequence of processor array dimensions for which a section subscript has been specified. Note that* $m' \geq n'$ *must be satisfied.*
3. *The dimension permutation given by* $\pi = (\pi_1, \ldots, \pi_{m'})$ *must be a permutation of* $(1, \ldots, m')$.
4. *The sequence of dimensions of R to which the distributed dimensions of A are mapped is given by* $dim_\mu = (p_1, \ldots, p_{m'}) = \pi(d'_1, \ldots, d'_{m'})$. *The set of dimensions for which a scalar subscript has been specified is denoted by* $dim_s = \{1, \ldots, m\} - \{p_1, \ldots, p_{m'}\}$.
5. *The distributed array dimensions* $d_1, \ldots, d_{n'}$ *are mapped to the processor dimensions* $p_1, \ldots, p_{m'}$ *(from left to right) such that for each* $j, 1 \leq j \leq n'$, *there exists a dimensional distribution function*
$$\delta_{R'_{p_j}}^{A_{d_j}} : \mathbf{I}_{d_j}^A \to \mathcal{P}(\bar{\mathbf{I}}_{p_j}^{R'}) - \{\phi\}.$$
6. *The distribution function* $\delta_{R'}^A$ *can be constructed from the dimensional distribution functions as follows: For each* $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{I}^A$*:*
$$\delta_{R'}^A(\mathbf{i}) = \{(r_1, \ldots, r_m) \in \bar{\mathbf{I}}^{R'} \mid \forall j, 1 \leq j \leq n', p_j \in dim_\mu : r_{p_j} \in \delta_{R'_{p_j}}^{A_{d_j}}(i_{d_j})\}$$
   For the $m - m'$ *processor dimensions* r_k, $k \in dim_s$, *for which the corresponding subscript* ss_k *is scalar, we have* $r_k = ss_k$. *If* $m' > n'$, *then the* $m' - n'$ *processor dimensions* $p_{n'+1}, \ldots, p_{m'}$ *are universally quantified.* ■

Composite distributions are illustrated in the following examples:

Example 2.4.5: Composite distributions

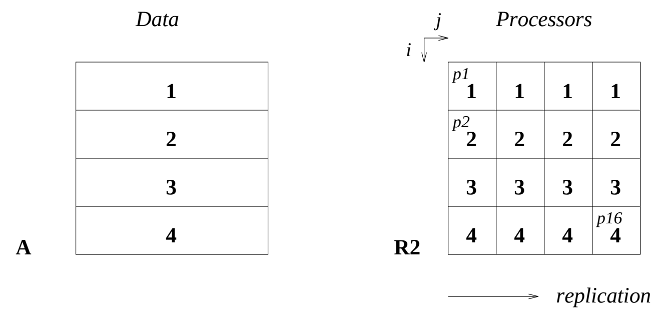
```
PROCESSORS R(2,10,10,10)
REAL, DIMENSION(10,10,10,10) :: A
DISTRIBUTED(:,BLOCK,:,BLOCK) TO R(1,1:10:2,3:3,1:5)(/3,2,1/) :: A
```

The distribution rank of A is $n' = 2$, the rank of the processor array section is $m' = 3$. The sequence of distributed dimensions is given by $dim_\delta = (2,4)$. The sequence of processor array dimensions to which the distributed dimensions are mapped is given by $dim_\mu = (4,3,2)$. Therefore, the 2^{nd} dimension of A is distributed over the 4^{th} dimension of R, and the 4^{th} dimension of A is distributed to the 3^{rd} dimension of R. Replication occurs over the the specified section of the 2^{nd} dimension of R. The extended index domain of the processor array section is given by $\bar{\mathbf{I}}^{R'} = [1:1, 1:10:2, 3:3, 1:5]$. The resulting distribution function is given by:

$$\delta_R^A(i_1,i_2,i_3,i_4) = \left\{(1,r_2,r_3,r_4) \mid r_2 \in \{1,3,5,7,9\}, r_3 \in \delta^{A_4}_{R'_3}(i_4), r_4 \in \delta^{A_2}_{R'_4}(i_2)\right\}$$

■

Example 2.4.6: Replication over one processor array dimension

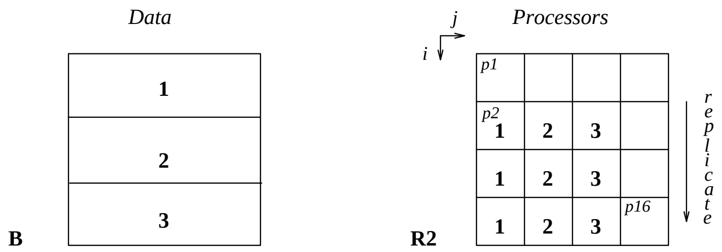



```
PROCESSORS :: R2(4,4)
REAL, DISTRIBUTED(BLOCK,:) :: A(N,N)
```

In this example the array A is distributed by BLOCK in the first dimension while the second dimension stays undistributed. The resulting 4 segments of array A are mapped to the first dimension of processor array R2 and replicated over the second dimension of R2 because the distribution rank of array A is 1. Under the assumption that 4 exactly divides N the distribution function of A is given by

$$\delta^A(i,j) = \left\{\left(\left\lceil \frac{i}{N/4}\right\rceil, r\right) \mid 1 \le r \le 4\right\},\ \forall\ i,j,\ 1 \le i,j \le N.$$

■



Example 2.4.7: Dimension permutation

```
PROCESSORS :: R2(4,4)
REAL, DISTRIBUTED(BLOCK,:) TO R2(2:,:3)(/2,1/) :: B(N,N)
```

This example illustrates the effect of a dimension permutation together with a processor array section. Because of the given dimension permutation array B's first dimension is partitioned into 3 segments according to the section extent of the second processor array dimension. These 3 segments are mapped to the second dimension of the processor array section and replicated over the first dimension. Under the assumption that 3 exactly divides N the distribution function of B is given by

$$\delta^B(i,j) = \left\{ \left(r, \left\lceil \frac{i}{N/3} \right\rceil \right) \mid 2 \leq r \leq 4 \right\}, \; \forall \; i,j, \; 1 \leq i,j \leq N.$$

■

All the intrinsic distribution functions described so far partition an array dimension into disjoint parts in such a way that each part can be represented by means of a regular section. Therefore these distributions are often referred to as regular distributions. Whereas regular distributions are well suited for many problems based on regular grids they usually are insufficient for applications that operate on unstructured or irregular meshes.

Indirect Distribution

Vienna Fortran provides indirect distributions to support applications which are characterized by irregular data distribution and access patterns. Indirect distributions are defined by means of mapping arrays and allow arbitrary mappings of array elements to processors. However, since each array element must be mapped to exactly one processor replication is not possible.

In Vienna Fortran 90 the intrinsic distribution function *INDIRECT* may be used to specify a distribution according to the elements of a mapping array. Mapping arrays must be integer arrays and may be associated with a distribution in the usual way. The distribution function *INDIRECT* may either be used as a dimensional distribution function to specify the distribution of a single array dimension or as a $n-m$ distribution function to specify the distribution of an n-dimensional array with respect to an m-dimensional processor array.

Example 2.4.8: Indirect distribution

```
PROCESSORS R2(4,2), R(8)

INTEGER, PARAMETER :: n1=2, n2=4, m1=2, m2=4, N=8, M=8
INTEGER, PARAMETER :: MAP1(N) = (/ 1,2,3,4,4,3,2,1 /)
INTEGER, PARAMETER :: MAP2(N,M) = RESHAPE((/                         &
         ((1,i=1,n1), (2,i=1,n2-n1), (5,i=1,N-n2), j=1,m1),          &
         ((3,i=1,n1), (4,i=1,n2-n1), (5,i=1,N-n2), j=1,m2-m1),       &
         ((6,i=1,n2), (7,i=1,N-n2),                j=1,M-m2) /),(/N,M/))

REAL, DISTRIBUTED(INDIRECT(MAP1),BLOCK) :: A(N,N)
REAL, DISTRIBUTED(INDIRECT(MAP2))       :: B(N,M)
```

Array A has an indirect distribution only in the first dimension. Array B is distributed according to the contents of array MAP giving the depicted distribution pattern illustrated in the right part of the above figure. ∎

Assume that we have an n-dimensional array A with index domain $\mathbf{I}^A$, a mapping array M with index domain $\mathbf{I}^M$, and an m-dimensional processor array R with index domain $\mathbf{I}^R$.

Case 1:

If *INDIRECT* is used as a dimensional distribution function, then the mapping array must be 1-dimensional and must have the same size as the corresponding array dimension. Correspondence between the indices of the array dimension and the elements of the mapping array is established by means of the Fortran array element order. Each element of the mapping array must represent a valid index of the associated index dimension of the corresponding processor array section. If the j-th dimension of A is distributed by $INDIRECT(M)$ with respect to the k-th dimension of R, then each index $\mathbf{i} \in \mathbf{I}^A_j$ is mapped to index $M(\mathbf{i}') \in \mathbf{I}^R_k$, where $\mathbf{i}'$ is determined according to the Fortran array element order. If $M(\mathbf{i}') \notin \mathbf{I}^R_k$, an error message will be issued.

Case 2:

If *INDIRECT* is used as a general distribution function, then the mapping array must be shape conform with the array to be distributed. Correspondence between the elements of the data array and the elements of the mapping array is established by means of the Fortran array element order. Each index $\mathbf{i} \in \mathbf{I}^A$ is mapped to index $M(\mathbf{i}') \in \mathbf{I}^{R'}$, where R' is given by $RESHAPE(R, (/SIZE(R)/))$. The correspondence between $\mathbf{i}$ and $\mathbf{i}'$ is established as above.

Indirect distributions are shown in Example 2.4.8.

2.4.2 Implicit Specification of Distributions

Vienna Fortran provides two different features to specify the distribution of one array in terms of the distribution of another array - alignment and distribution extraction. Distribution by alignment is based on a relationship between the index domain of two arrays. Distribution extraction is a way to distribute an array by employing the same distribution function which was used to distribute another array.

2.4.2.1 The ALIGNED Attribute

Alignment is a method to define the distribution of an array B implicitly by specifying for each element that it should be distributed to the same processor(s) as a certain element (or section) of an array A. Array B is called **alignee** and array A is called the **source array** of the alignment.

Alignment between statically distributed arrays has to be specified by means of the *ALIGNED* attribute. This can be done within the type declaration statement or by using the *ALIGNED statement* in the specification part of a program unit. An alignment relationship between two statically distributed and statically allocated arrays that is established by means of the *ALIGNED* attribute is invariant within the scope of the declaration, and therefore called **static alignment**. Alignment in the context of dynamically distributed arrays is discussed in Section 2.4.4.

Vienna Fortran 90 distinguishes three forms of alignment specifications: **element-based alignment**, **section-based alignment**, and **functional alignment**. Element-based alignment specifies for each element of the alignee to which element (or section) of the source array it should be mapped. As will be shown below this includes collapsing and permutation of dimensions and replication. By means of section-based alignment, it is only possible to align the alignee with a conformable section of the source array. This allows for embedding the alignee into the source array, possibly using offsets and strides, but without allowing any replication, collapsing, or permutation of dimensions. Arbitrary alignments can be specified by means of functional alignment using mapping procedures as explained in Section 2.4.3.

The *ALIGNED* attribute consists of the keyword *ALIGNED* followed by an *alignment specification*. In case of an element-based alignment the *alignment specification* consists of three components: an *alignee-specification*, followed by the keyword *WITH* and an *alignment-source-specification*. The alignee-specification consists of the (optional) name of the *alignee* and either a reference to an alignment function or a list of bound variables or colons. The *alignment-source-specification* must designate an array element or an array section. In case of section-based alignment the *alignee-specification* and the keyword *WITH* may be omitted.

Element-Based Alignment

Let us first consider an *ALIGNED* attribute of the following form, which defines an element-based alignment:

ALIGNED(B($bs_1, \ldots, bs_n$) WITH A($as_1, \ldots, as_m$))

where $\mathbf{I}^B = [L_1^B : U_1^B, \ldots, L_n^B : U_n^B]$ denotes the index domain of the alignee B and $\mathbf{I}^A = [L_1^A : U_1^A, \ldots, L_n^A : U_m^A]$ denotes the index domain of the align-source array A, respectively.

Each bs_i, $1 \leq i \leq n$, is either

- a **bound variable** whose scope is limited to the alignment specification, and which ranges over the $i-th$ dimension of the index domain associated with B. It is implicitly declared of type integer, and any program variable with the same name is hidden until the end of the alignment specification.
  or
- a colon, which is implicitly replaced by a newly generated bound variable and for which the same rules as those outlined above hold.

Each as_j, $1 \leq j \leq m$, is either

- a *subscript* that does not contain a bound variable,
  or
- a *subscript triplet* which must not contain a bound variable,
  or
- an *align-subscript* that contains exactly one of the bound variables, say b_i. Then, no other *align-subscript* may contain b_i. as_j must be a specification expression that is linear in b_i and contains operators no other than "+", "-", and "*". Furthermore, the Fortran 90 intrinsic functions *LBOUND, UBOUND*, and *SIZE* may be applied.

The semantics of such an alignment specification is defined as follows:

1. Let $dim_\alpha = (d_1, \ldots, d_l)$ denote the sequence of aligned dimensions[11]. Let furthermore c denote the number of collapsed dimensions[12] of B and r the number of replicated dimensions[13] of A. Note that $n = l + c$ and $m = l + r$ must be satisfied.
2. Let $dim_\mu = (d'_1, \ldots, d'_l)$ denote the sequence of dimensions of array A to which the aligned dimensions of B are mapped. For each $i, 1 \leq i \leq l$, dimension d_i of B is mapped to dimension d'_i of A by a dimensional alignment function $\alpha_{A_{d'_i}}^{B_{d_i}}$.

[11]These are those dimensions for which a *bound-variable*, say i, is specified in the corresponding dimension of the *alignee-specification* which is associated with a non-replicated dimension of A (i.e. i also appears in the alignment-source-specification).

[12]These are those dimensions of B for which a colon is specified or a bound variable which does not occur in the alignment-source-specification.

[13]These are the dimensions of A where a subscript triplet occurs.

3. The alignment function α_A^B can be determined as follows:
   For each $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{I}^B$

$$\alpha_A^B(\mathbf{i}) = \{(j_1, \ldots, j_m) \in \mathbf{I}^A \mid \forall k, 1 \leq k \leq l : j_{d'_k} \in \alpha_{A_{d'_k}}^{B_{d_k}}(i_{d_k})\}$$

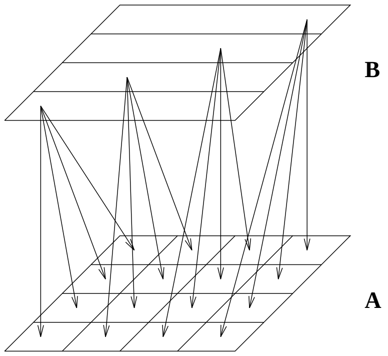
   For $m > l$ the r replicated dimensions, given by $\{1, \ldots, m\} - \{d'_1, \ldots, d'_l\}$, are universally quantified.

4. If A, B, δ_R^A, and α_A^B are given as above, then δ_R^B can be computed as shown in Definition 2.6.

Note that for the specification of element-based alignment a colon is used to express both collapsing and replication. If a colon appears in the *alignee-specification*, then the corresponding dimension is collapsed. If a colon appears in the *align-source-spec*, then replication occurs over the corresponding source array dimension.

Example 2.4.9: Collapsed and replicated alignment

```
ALIGNED(B(I,:) WITH A(:,I)) :: B(N,M1)
REAL                        :: A(M2,N)
```



In this example the first dimension of alignee B is aligned with the second dimension of the source array A. Since in the second dimension of the alignee a ':' is specified, this dimension is collapsed. Replication occurs over the first dimension of A, because a ':' is specified in the first dimension of the align-source specification. The alignment function is defined as follows:

$$\alpha_A^B(i_1, i_2) = \{(j_1, j_2) \in \mathbf{I}^A \mid j_2 \in \alpha_{A_2}^{B_1}(i_1)\} = \{(j_1, i_2) \mid 1 \leq j_1 \leq M2\}.$$

■

Section-Based Alignment

The second form of the *ALIGNED* attribute, called *section-based alignment*, is provided in order to align the alignee with a conformable section of the source array. Alignment offset, alignment strides, or embedded alignments therefore can be specified without the need for introducing bound variables as it would be the case if the element-based form of the alignment is used. However, by means of section-based alignment it is not possible to specify replication, collapsing and dimension permutations.

Example 2.4.10: Section-based versus element-based alignment

```
REAL, DIMENSION(10,20,10,20)                     :: A
REAL, ALIGNED(A(2,10:16:3,1,10:1:-2))            :: B(3:5, 0:4)
REAL, ALIGNED(B(I1,I2) WITH A(2,I1*3+1,1,10-I2*2)) :: B(3:5, 0:4)
```

The ALIGNED attributes in the two declaration statements for array B are equivalent.

■

Consider a *section-based alignment* of the following form:

$$\texttt{ALIGNED(A(}as_1, \ldots, as_m\texttt{))} \;\; \texttt{::} \;\; \texttt{B(}L_1^B : U_1^B \ldots, L_n^B : U_n^B \texttt{)}$$

with associated index domains $\mathbf{I}^A = [L_1^A : U_1^A, \ldots, L_n^A : U_m^A\,]$ and $\mathbf{I}^B = [L_1^B : U_1^B, \ldots, L_n^B : U_n^B]$, respectively.

Each as_j, $1 \leq j \leq m$, $m \geq n$, is either a *subscript*, a *subscript triplet*, or a *vector subscript*. The source array section $A(as_1, \ldots, as_m)$ must must be conformable with the alignee B. This implies that exactly n as_i must denote *subscript triplet* and all others must be scalar subscripts.

The effect of this alignment specification is such that each element of the alignee B is mapped to its corresponding element of the source array section $A(as_1, \ldots, as_m)$. This correspondence is defined by the Fortran 90 array element order.

Note that by using vector-subscripts it is possible to specify an indirect alignment. Each element of the indirection vector must represent a valid index in the corresponding dimension of the source array.

Assume that each as_i that represents a *subscript triplet* is of the form $[l_i : u_i : s_i]$. The above section-based alignment attribute is then equivalent to an element-based alignment attribute of the following form:

$$\texttt{ALIGNED(B(}I_1, \ldots, I_n\texttt{) WITH A(}as'_1, \ldots, as'_m\texttt{))}$$

with

$$as'_i = \begin{cases} (I_j - L_j^B) * s_i + l_i & \text{iff } as_i \text{ denotes the } j\text{-th } \textit{subscript triplet} \\ as_i & \text{iff } as_i \text{ is a scalar subscript} \end{cases}$$

Example 2.4.11: Alignment

```
REAL, DISTRIBUTED(BLOCK,CYCLIC)        :: S(100,100)
REAL, DISTRIBUTED(BLOCK)               :: S1(100)
INTEGER, PARAMETER, DIMENSION(100)     :: X = (/ .... /)

! Perfect Alignment
REAL, ALIGNED(A(I,J) WITH S(I,J))      :: A(100,100)      ! (a)
REAL, ALIGNED(S)                       :: A(100,100)      ! (a')

! Offsets
REAL, ALIGNED(B(I,J) WITH S(I+1,J+1))  :: B(0:99,0:99)    ! (b)
REAL, ALIGNED(S)                       :: B(0:99,0:99)    ! (b')

! Strides
REAL, ALIGNED(C(I,J) WITH S(2*I,2*J-1)) :: C(50,50)       ! (c)
REAL, ALIGNED(S(2::2,::2))             :: C(50,50)        ! (c')

! Permutation of Axes
REAL, ALIGNED(D(I,J) WITH S(J,I))      :: D(100,100)      ! (d)

! Collapsing
REAL, ALIGNED(E(I,:) WITH S1(I))       :: E(100,100)      ! (e)

! Embedding
REAL, ALIGNED(F(I) WITH S(I,5))        :: F(100)          ! (f)
REAL, ALIGNED(S(:,5))                  :: F(100)          ! (f')

! Replication
REAL, ALIGNED(G(I) WITH S1(:))         :: G(100)
REAL, ALIGNED(G(:) WITH S1)            :: G(100)          ! (g)
REAL, ALIGNED(G(:) WITH S1(:))         :: G(100)

REAL, ALIGNED(H(I) WITH S(I,:)         :: H(100)          ! (h)

! Partial Replication
REAL, ALIGNED(P(I) WITH S(I,::2)       :: P(100)          ! (i)

! Indirect Alignment
REAL, ALIGNED(WITH S1(X))              :: Q(100)          ! (j)
```

The declarations in Example 2.4.11 have the following semantics:
(a) Element A(I,J) is always on the same processor(s) as element S(I,J).
(b) Element B(I,J) is always on the same processor(s) as element S(I+1,J+1).
*(c) Element C(I,J) is always on the same processor(s) as element S(2*I,2*J).*
(d) Element D(I,J) is always on the same processor(s) as element S(J,I).
(e) The I-th *row of E is placed on the same processor(s) as S1(I).*
(f) Array F is distributed to the same processor(s) as the 5-th *row of S.*
(g) All three examples have the same meaning: every processor that has an element of S1 gets the whole array G.

(h) A copy of H is aligned with every column of S; or in other words: element G(I) is aligned with every element of the i-th row of S.
(i) Array P is aligned with every other column of S.[14] *As a consequence, element P(I) will be allocated on the same processor(s) as element S(I,2*K-1),* $1 \leq K \leq 50$.
(j) Array Q is aligned with an irregular section of array S1. Since Q is statically distributed and statically allocated, the vector X must be an array constant.

The examples (a') - (d') and (f') are equivalent alignment specifications expressed by means of section-based alignment *rather than element-based alignment.* ■

As already mentioned, an alignment relationship between two statically distributed and statically allocated arrays, which has been established at the time of declaration by means of the *ALIGNED* attribute, remains invariant within the scope of declaration. However, in the context of dynamically allocated arrays the alignment relationship may be broken, since the index domains of the arrays may be modified as the result of the execution of an allocate statement. This is discussed in more detail in Section 2.4.5. In order to enable the user to specify sets of dynamically distributed arrays that have a run-time invariant alignment relationship, Vienna Fortran provides the *CONNECT* attribute which is discussed in Section 2.4.4.1.

Generally, at the time an object is allocated it may be aligned only with an object that is already allocated.[15] This has the following consequences:

1. A statically allocated array must not be aligned with a dynamically allocated array, since the index domain of an allocatable array (or a pointer) is undefined at the time of declaration.
2. An array that has the *ALLOCATABLE* attribute may appear as a source array only if its allocation status is *ALLOCATED*.
3. An array that has the *POINTER* attribute may appear as a source array only if its pointer association status is defined.
4. An array that has the *DYNAMIC* attribute may appear as an align source within the *ALIGNED* attribute only if it has an initial distribution (see Section 2.4.4.2).

Multilevel Alignment

In order to alleviate the alignment between groups of arrays, Vienna Fortran 90 provides the concept of multilevel alignment by allowing that an alignment source array may appear also as an alignee. For example the distribution of an array *C* may be defined by aligning it with an array *B* which in turn is aligned with an array *A*. An array is called **alignment basis** if its distribution is specified directly by means of the *DISTRIBUTED* attribute. Therefore, an alignment relationship established between a set of arrays and the corresponding alignment basis can be represented by means of an **alignment tree** with the alignment basis as root.

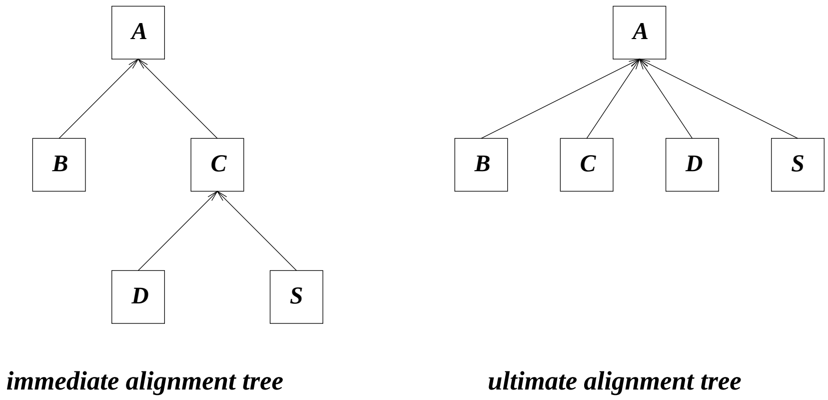
[14] Note that this cannot be expressed in HPF.
[15] Dynamically allocated arrays are created whenever a corresponding *ALLOCATE* statement is executed. Statically allocated arrays are created at the time their declaration is evaluated.

An alignment tree of height n is called

- **degenerate** iff $n = 0$
- **ultimate** iff $n = 1$
- **immediate** iff $n > 1$

Assume that an array B is aligned with an array A. If A is an *alignment basis*, then we say B is **ultimately aligned** with A, otherwise B is **immediately aligned** with A. Immediate alignment can always be transformed into ultimate alignment and thus any alignment tree of height $n > 1$ can be transformed into an alignment tree of height 1. All objects in an alignment tree therefore are ultimately aligned with the alignment basis. For instance array D in example 2.4.12 is immediately aligned with C but ultimately aligned with A.

Example 2.4.12: Multilevel alignment




```
DISTRIBUTED(BLOCK)         :: A(2*N)
ALIGNED(WITH A(1:N:2))     :: B(0:N-1)
ALIGNED(WITH A(2:N+1))     :: C(N)
ALIGNED(D(I) WITH C(I+2)) :: D(0:N-2)
ALIGNED(WITH C(::2))       :: S
```

In this example B and C are ultimately aligned with A, whereas D and S are immediately aligned with C. As shown in the above figure the immediate alignment tree of height 2 can be transformed into an ultimate alignment tree. The above alignment specifications of D and S are therefore equivalent to

```
ALIGNED(WITH A(3:N+1))   :: D(0:N-1)
ALIGNED(WITH A(2:N+1:2)) :: S
```

■

2.4.2.2 Distribution Extraction

If one array is to be distributed in the same way as another array, then its distribution can be defined by means of distribution extraction.

A **distribution extraction** is specified in the form

$$= A[.dim]$$

where A is an array followed by an optional dimension specifier. At the time of evaluation of this construct, A must be associated with a well-defined distribution, say δ^A. If dim is specified, δ^A must be composite, and dim an integer constant expression with $1 \leq dim \leq rank(A)$.

If the dimension specifier is not present, then the distribution extraction yields the distribution type associated with δ^A. Otherwise, it yields the distribution type of the dimension of A determined by the value of dim. A relationship between statically distributed arrays that is established by means of distribution extraction is invariant within their scope of declaration.

Example 2.4.13: Distribution extraction

```
REAL, DISTRIBUTED(BLOCK,CYCLIC) :: A(N,N)
REAL, DISTRIBUTED(=A)           :: B(M,M)
REAL, DISTRIBUTED(=A.2,:)       :: C(N,M)
```

In this example the declarations of B and C are equivalent to

```
REAL, DISTRIBUTED(BLOCK,CYCLIC) :: B(M,M)
REAL, DISTRIBUTED(CYCLIC,:)     :: C(N,M)
```

■

Note that if the distribution of an array B is specified by means of distribution extraction with respect to an array A then $\delta^B = \delta^A$ only if both arrays have the same index domain and are distributed with respect to the same processor array.

If a distribution extraction occurring in the *DISTRIBUTED* attribute refers to a dynamically distributed array, say A, then A must have an initial distribution specified. Furthermore, a distribution extraction must not refer to a dummy array with inherited distribution (see Section 2.5.3.2). Distribution extraction in the context of dynamically distributed arrays is discussed in Section 2.4.4. Distribution extraction involving components of derived types is discussed in Section 2.4.6.

2.4.3 Mapping Procedures

Vienna Fortran [139] provides user defined distribution and alignment functions. We have unified and generalized these concepts by introducing **mapping procedures**. By means of mapping procedures it is possible to specify arbitrary distributions or alignments. Furthermore, mapping procedures may be used to specify the work distribution of *FORALL* loops. The activation of a mapping procedure results in the computation of a distribution or an alignment as explained in Section 2.2.1. If a mapping procedure is activated in the context of a *FORALL* loop, it determines a mapping of loop iterations to processors. Mapping procedures may have arbitrary arguments as a normal Fortran procedure. However, all the arguments implicitly have the INTENT(IN) attribute. Apart from the computation of a distribution or an alignment these functions must not produce any side effects.

A mapping procedure has two implicit arguments, the **from-domain** and the **to-domain**[16] which can be thought of as arrays in the sense of Fortran with the difference that they do not have a type and do not occupy any storage.

Example 2.4.14: User defined distribution functions

```
PROCESSORS R(4,4)
REAL, DISTRIBUTED(HPF_BLOCK,HPF_BLOCK) TO R(:,4:1-1) :: A(N,N)
...
MAPPING HPF_BLOCK
   FROM F(:) TO T(:)

   INTEGER N,M

   M = CEILING(REAL(SIZE(F))/REAL(SIZE(T)))   ! block size
   N = SIZE(F)/M                              ! # of blocks of size M

   DO I = 1, N                                ! map blocks of size M
      F((I-1)*M+1 : I*M) -> T(I)              ! index mapping statement
   END DO

   F(NM*M+1,:) -> T(N+1)                      ! map remaining elements

END MAPPING HPF_BLOCK
```

This is an example of using a mapping function in order to specify a variant of the BLOCK distribution. In the first incarnation the to-domain T is associated with the indices $\{1, 2, \ldots, 4\}$ *and in the second incarnation T is associated with the indices* $\{4, 3, \ldots, 1\}$. ∎

A reference to a mapping procedure can occur in the context of a distribution specification, alignment specification, or work distribution specification.

- If a mapping procedure is referenced within a distribution specification, the from-domain is implicitly associated with the index domain (dimension) of the array (dimension) to be distributed and the to-domain is associated with the index domain (dimension) of the processor array (dimension) to which the distribution refers.

[16] If a mapping procedure is used as a distribution function, the to-domain may be omitted. In this case, $\$P$ is regarded as the to-domain.

- If a mapping procedure is referenced in the context of an alignment specification, the from-domain is associated with the index domain (dimension) of the alignee and the to-domain is associated with the index domain (dimension) of the alignment-source array.
- If a mapping procedure is referenced in the work distribution part of a *FORALL* loop, then the from-domain is associated with the index domain defined by the loop iteration space and to-domain is associated with the index domain (dimension) of the corresponding processor array.

If the rank of the from-domain is n and the rank of the to-domain is m, then we speak of an n-m **mapping procedure**.

Example 2.4.15: User defined alignment functions

```
REAL, DISTRIBUTED(BLOCK,:)     :: A(100,100)
REAL, ALIGNED(DIAGONAL WITH A) :: B(100,100)
...
MAPPING DIAGONAL
   FROM F(:,:) TO T(:,:)

   DO I = 1, SIZE(T,1)
      F(I,:) -> T(I,I)
   END DO
END MAPPING DIAGONAL
```

In this example the rows of array B are aligned with the main diagonal of array A. Therefore, each row A(I,:) will be allocated on the same processor as element B(I,I), $1 \leq I \leq 100$. ∎

Mapping procedures must realize an index mapping as described in Definition 4, i.e. each index of the from-domain must be mapped to one or more indices of the to-domain. This mapping has to be specified by means of the *index mapping statement* which may have one of the following two forms:

1. *fref* `->` *tref*
2. *fref* `MAP(`*mts*`)` `TO` *tref*

where *fref* and *tref* designate either an element or a section of the from-domain and to-domain, respectively. The *mapping-type specification* (*mts*) is a reference to one or more mapping procedures (user-defined or intrinsic).

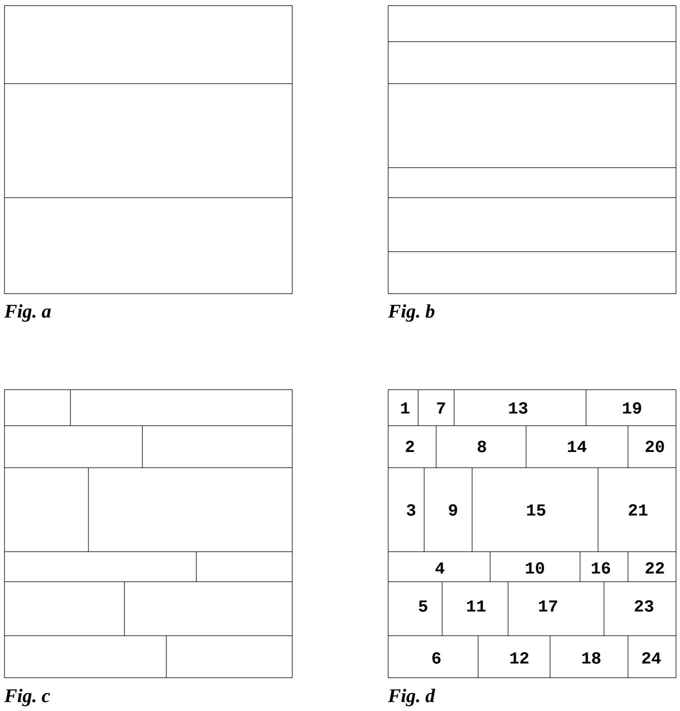
Lets assume an index mapping statement of the first form:

fref `->` *tref*

where *fref* and *tref* designate either an element or a section of the from-domain and to-domain, respectively. Assume furthermore that the corresponding mapping function is referenced within a distribution specification. Let $\mathbf{I}^A$ denote the index domain of the array (dimension) to be distributed and $\mathbf{J}^R$ denote the index domain of the corresponding processor array (dimension). Let furthermore $\mathbf{I}' \subseteq \mathbf{I}^A$ denote the set of indices assoicated with $fref$ and $\mathbf{J}' \subseteq \mathbf{J}^R$ the indices associated with $tref$ The effect of such an index mapping statement is as follows:

for each $\mathbf{i} \in \mathbf{I}'$
  for each $\mathbf{j} \in \mathbf{J}'$
    $\delta_R^A(\mathbf{i})$:= $\delta_R^A(\mathbf{i}) \cup \{\mathbf{j}\}$
  endfor
endfor

If the second form of the index mapping statement is used, the mapping is carried out according to the referenced mapping procedure or according to the specified intrinsic function.




```
PROCESSORS R(6,4)
...
DISTRIBUTED(MRD(A,(/3,2/),(/2,2/))) :: A(N,N)
```

Figure 2.2: Multiple recursive distribution

Mapping procedures also can be recursive. This feature is in particular very useful to specify recursive distributions such as *binary or multiple recursive distributions* [17, 106] which are often

Example 2.4.16: Multiple recursive distributions

```
RECURSIVE MAPPING MRD(A,PF1,PF2)
   FROM IA(:,:) TO IR(:,:)
   REAL A(:,:)                         ! data array
   INTEGER, DIMENSION(:) :: PF1,PF2    ! prime factor lists
   INTEGER, ALLOCATABLE  :: bnds(:)    ! bounds of submatrices

   IF (SIZE(PF1) > 0) THEN             ! partition the first dimension
      bnds => ...                      ! determine bounds of submatrices
      DO I=1, SIZE(bnds)-1
         IA(bnds(I):bnds(I+1),:)                                      &
           MAP(MRD(A(bnds(I):bnds(I+1),:),PF1(2:),PF2)) TO            &
         IR((I-1)*SIZE(IR,1)/PF1(1):I*SIZE(IR,1)/PF1(1),:)
      END DO
   ELSE
      IF (SIZE(PF1) > 0) THEN          ! partition the second dimension
         bnds => ...                   ! determine bounds of submatrices
         DO I=1, SIZE(bnds)-1
            IA(:,bnds(I):bnds(I+1))                                   &
               MAP (MRD(A(:,bnds(I):bnds(I+1)),PF1,PF2(2:)) TO        &
            IR(:,(I-1)*SIZE(IR,2)/PF2(1):I*SIZE(IR,2)/PF2(1))
         END DO
      ELSE
         IA -> IR                        ! map submatrix
      END IF
   END IF
END MAPPING MRD
```

This is an example of a multiple recursive distribution of a 2-dimensional matrix with respect to a 2-dimensional processor array. Assume that we have $p * q$ processors and assume that the prime factor decomposition of p and q is given by $p = p_1.p_2 \ldots .p_r$ and $q = q_1.q_2 \ldots q_s$, $r, s \geq 1$. The prime factors are stored in the arrays PF1 and PF2, respectively. The multiple recursive distribution is carried out in $r + s$ stages as follows. In the first r stages the first dimension of the array is partitioned recursively into m submatrices. In stage 1 the first dimension of IA is divided into p_1 parts such that the load is balanced equally among the resulting submatrices (i.e each submatrix contains the same number of non-zero elements). For each of the submatrices the mapping procedure is invoked recursively, and each of the submatrices is decomposed subsequently into p_2 submatrices, and so on. In stages $r + 1$ to $r + s$ the same technique is applied to divide the second dimension according to the prime factors contained in PF2. In Figure 2.2 it is assumed that $m = 6$ and $n = 4$. Fig. a. and b. show the decomposition of the first dimension and Fig. c. and d. the partition of the second dimension. The numbers in Fig.d. indicate to which processors (with respect to the implicit processor array $P) the resulting segments are mapped. ∎

used to achieve good load balancing for sparse matrix computations. In the following we show how multiple recursive distributions as proposed in [106] can be specified by means of a recursive mapping procedure as shown in Example 2.4.16. A multiple recursive distribution with respect to a 6×4 processor array is visualized in Figure 2.2.

2.4.3.1 Expressing Alignment via User Defined Distribution Functions

The distribution type of an object is determined by one ore more distribution functions together with the list of evaluated arguments. If the distribution of an array is specified implicitly by means of alignment, the resulting distribution can always be described directly (i.e as a one-level mapping) by means of a mapping procedure. Thus, the distribution type of an aligned object is determined by a mapping procedure together with its evaluated arguments. In the following we show how the distribution of an alignee can be expressed by means of a user defined distribution function (i.e mapping procedure) based on the alignment function and the distribution of the alignment source.

Assume that the distribution of an alignee B is specified by means of a user defined alignment function $UDAF$ with respect to a source array A and assume that A is distributed somehow with respect to the default processor array $\$P$. The resulting distribution of B with respect to $\$P$ can be described by means of a user defined distribution function $UDDF$ as shown in Example 2.4.17.

Example 2.4.17: Transforming alignment functions into distribution functions

```
ALIGNED(UDAF WITH A) :: B(M)

! is equivalent to

DISTRIBUTED(UDDF(UDAF,A)) :: B(M)
```

```
MAPPING UDDF(UDAF,A)
  FROM B(:)
  EXTERNAL MAPPING M
  REAL A(:,:)

  DO I=1, SIZE(B)
     SET => EVAL_MAPPING(B(I),A,M)
     DO K=1, SIZE(SET)
        B(I) -> $P(OWNERS(A(K)))
     END DO
  END DO
END MAPPING
```

This is a sketch of how user defined alignment functions can be expressed by means of user defined distribution functions. The intrinsic function $EVAL\_MAPPING$ takes an element $B(I)$ of the from-domain, an array and a mapping procedure M as arguments and returns a pointer to a rank one array which contains the indices of all elements of A that are associated with $B(I)$ by means of the mapping procedure M. The intrinsic function $OWNERS$ takes as argument an array element and returns a rank one array containing the indices of all processors in $\$P$ that own the array element. ∎

2.4.4 Dynamic Data Distribution

So far we have only considered statically distributed arrays, which have one fixed distribution type within their scope of declaration. An array whose distribution type may be changed at runtime is called a **dynamically distributed array** and must be declared with the *DYNAMIC* attribute. Distributions for such arrays are determined at run-time and may change during the execution of the program as the result of a procedure call or the execution of a *DISTRIBUTE* or *ALIGN* statement.

Dynamic data distribution is important in order to support applications where the distribution of an array needs to be changed for different program phases in order to achieve good performance. However, unrestricted use of dynamically distributed arrays may have a severe impact on the compilation process and the efficiency of the resulting program. Therefore, Vienna Fortran provides several language constructs that supply the compiler with additional information (i.e about the range of possible distributions an array may have at run-time) in order to generate more efficient target code. For this purpose four additional attributes are provided, i.e. *CONNECT attribute*, *RANGE attribute*, *NOTRANSFER attribute*, and the *RESTORE attribute*.

2.4.4.1 Array Connection

It is often desirable to combine arrays that should always be distributed together. In the case of statically distributed arrays this can be achieved by constructing align sets or by means of distribution extraction. A relationship between the distribution of statically distributed and statically allocated arrays caused by distribution extraction or alignment stays invariant within the scope of declaration.

Such a relationship between two dynamically distributed arrays, is usually not maintained during the execution of the program, i.e. it may be destroyed if one of the arrays is associated with a new distribution (see Example 2.4.21). In order to define sets of dynamically distributed arrays with an invariant relationship between their distributions, Vienna Fortran provides the *CONNECT* attribute. It consists of the keyword *CONNECT* followed by an alignment specification or a distribution extraction.

Depending on whether the *CONNECT* attribute has been specified dynamically distributed arrays can be divided into two classes; **secondary arrays** which must have the *CONNECT* attribute, and **primary arrays**, which must not be connected to other arrays and which may be the only objects of a distribute statement. A secondary array must either be connected to a primary array or to another secondary array. Secondary arrays are implicitly associated with a new distribution whenever the distribution of the corresponding primary array is changed. Note that each secondary array must be connected with exactly one primary array, i.e. it is not allowed to connect different array dimensions to different primary arrays by means of distribution extraction.

Assume that a secondary array B is connected with an array A. As in the case of alignment we say that B is **ultimately connected** with A if A is a primary array (i.e. does not have the CONNECT attribute) otherwise B is **immediately connected** with A. The relationship between a set of arrays that is established by means of the *CONNECT* attribute can be represented by means of a **connect tree**. As with alignment trees (page 42), a connect tree of height $n > 1$ can always be transformed into a tree of height 1.

The set of arrays consisting of a primary array, say A, and all secondary arrays that are ultimately connected to it is called **connect set** of A, denoted by $\mathcal{C}(A)$. The connect attribute introduces an equivalence relation in the set of dynamically distributed arrays, each connect set representing an equivalence class. This bundling mechanism allows to define groups of dynamically distributed arrays and to define a relationship[17] between the distributions of these arrays which is invariant during runtime. Array connection is shown in Example 2.4.18.

Example 2.4.18: Connect sets

```
REAL, DYNAMIC                              :: A(N,N), F(N,N)
REAL, DYNAMIC, CONNECT(=A)                 :: B(N,N), C(M,M)
REAL, DYNAMIC, CONNECT(D(I,J) WITH C(J,I)) :: D(N,N)
REAL, DYNAMIC, CONNECT(=D.1,=F.1)          :: E(N,N)          ! illegal
```

All arrays in this example are dynamically distributed. A and F are primary arrays all others are secondary arrays since they are specified with the CONNECT *attribute. The connect set of A consists of 4 arrays: $\mathcal{C}(A) = \{A, B, C, D\}$. Within the scope of this declaration B and C always have the same distribution type as A, and D is always aligned in a transposed fashion to A. The connect attribute for E is illegal since it specifies a connection with respect to different primary arrays A and F.* ∎

Whenever a primary array is associated with a new distribution all arrays in its connect set may have to be redistributed so that the relationship caused by the connection is maintained. Note that secondary arrays must not be the objects of a distribute or align statement. They may be associated with a new distribution only by redistributing the corresponding primary array. As a consequence, if a secondary array is passed to a procedure where it is redistributed, its distribution might have to be restored on exit in order to keep the relationship between its distribution and that of the corresponding primary array invariant. Note that an array to which other arrays are connected (i.e a primary array) must not be the object of an *ALIGN* statement.

2.4.4.2 Initial Distribution

For primary arrays, an **initial distribution** may be specified by means of the *DISTRIBUTED* attribute or *ALIGNED* attribute. Usually an array is associated with the initial distribution whenever an allocation instance is initiated. A dynamically distributed array without an initial distribution must not be accessed unless it has been associated with a distribution.

If an initial distribution is specified implicitly with respect to a dynamically distributed array, say A, it refers to the initial distribution of A. An initial distribution is evaluated when the declaration is evaluated. Allocatable arrays which are declared with an initial distribution are

[17] This refers to ultimate alignment. An immediate alignment relationship may be broken, for example if allocatable arrays are involved.

associated with this distribution each time an allocation instance is initiated, iff a distribution or alignment specification is not explicitly specified in the corresponding allocate statement (see Section 2.4.5).

Example 2.4.19: Initial distributions

```
REAL, DYNAMIC, DISTRIBUTED(BLOCK)     :: A(1000)
REAL, DYNAMIC, ALIGNED(WITH(A(:100))) :: B(100)
REAL, DYNAMIC, DISTRIBUTED(=A)        :: C(500)
```

For all three arrays in this example an initial distribution is specified. The arrays are associated with their initial distribution when they are created. Note that the distribution relationship established between B and A, and C and A respectively, usually will not stay invariant during runtime. ■

2.4.4.3 Distribution Range

A **distribution range** for a dynamically distributed array A specifies the set of all distribution types (or a superset thereof) which can be associated during runtime with A. A distribute statement for an array for which a distribution range is specified must respect the restrictions imposed by this attribute.

A distribution range specification is identical to a distribution specification with the difference that the "*" can be used as a wild-card symbol in any place where either a distribution specification or an argument for a distribution function may occur, with the meaning that it allows any legal substitution. Note that the range attribute only may be specified for dynamically distributed primary arrays and dummy arrays with assumed distribution (see Section 2.5.3.2). A distribution range is evaluated at the time the declaration is evaluated.

The default rule for the substitution of the primary processor array does not apply for distribution specifications occurring in the range attribute. If a distribution specification occurring in the *RANGE* attribute is specified without a processor reference, any processor set may be associated with the array.

Example 2.4.20: Distribution range

```
REAL, DYNAMIC, RANGE((BLOCK,:),(:,BLOCK),(CYCLIC(*),*)) :: A(N,N),B(N,N)
REAL, DYNAMIC, RANGE(=A) :: C(M,M)
```

The dynamically distributed arrays A and B in the above declaration may either be distributed by BLOCK in the first dimension and undistributed in the second dimension and vice versa, or CYCLIC distributed with any legal block length in the first distribution and any distribution in the second dimension (including ":"). The distribution extraction used in the RANGE *attribute of array C declares the distribution range of C to be the same as for A.* ■

Furthermore, if two arrays should be associated with the same distribution range the *RANGE* attribute may be specified by means of **distribution range extraction** which is syntactically similar to distribution extraction.[18] This is shown in Example 2.4.20.

2.4.4.4 Redistribution and Realignment

A dynamically distributed array may be associated with a new distribution by executing a *DISTRIBUTE* or *ALIGN* statement. All objects of a distribute statement must be primary arrays (i.e. they must not have the *CONNECT* attribute). An object of an align statement must not have any other objects connected to it.

For all objects of a distribute statement the corresponding distribution specification is evaluated and applied to it resulting in a new distribution.[19] Moreover, all secondary arrays contained in the connect set of an array occurring in the *distribution-list* also get associated with a new distribution according to the relationship defined in the *CONNECT* attribute. If the optional *STAT=* specifier is present, successful execution of the distribute statement causes the *stat-variable* to become defined with a value of zero. If an error condition occurs, the value of the *stat-variable* will be $\neq 0$.

Example 2.4.21: The *DISTRIBUTE* and the *ALIGN* statement

```
REAL, DYNAMIC, DISTRIBUTED(BLOCK,:) :: A(N,N)
REAL, DYNAMIC, CONNECT(=A)          :: B(N,N),C(N,N)
REAL, DYNAMIC, DISTRIBUTED(=A)      :: D(N,N)

DISTRIBUTE (A(:,BLOCK),NOTRANSFER(A,C))  ! (s1)

ALIGN (D(I,J) WITH A(J,I))               ! (s2)
```

At the beginning of the execution of the program unit containing the above declarations all 4 arrays are associated with the same distribution. The execution of the distribute statement changes the distribution type of A from (BLOCK,:) to (:,BLOCK). Since arrays B and C are connected to A, their distribution types are changed to (:,BLOCK) as well. The NOTRANSFER attribute has the effect that for A and C only the access-function gets changed, but no physical transfer of data takes place. As a consequence, the contents of A and C are undefined after the execution of the distribute statement. Since array D's distribution stays unchanged, the relationship between the distribution of A and D specified in the declaration is no longer valid after statement `s1`. *Statement* `s2` *is an example of an* ALIGN *statement which has the effect that each element* $D(I,J)$ *is allocated on those processors which own* $A(J,I)$.

■

The *NOTRANSFER* attribute is provided in order to specify that a redistribution or realignment of an array can be accomplished without the need of keeping its current values. This

[18] In this case, the dimension qualifier must not be present.

[19] In contrast to *dist-specs* or *align-specs* occurring in declarations, any expression occurring in a *dist-spec* or *align-spec* of a distribute or align statement may be an arbitrary integer expression.

enables the compiler to realize the new distribution by changing the access function of the corresponding object only. Any physical transfer of data during the redistribution can be avoided. If the *NOTRANSFER* attribute is specified within the distribute statement, it may be followed by a list of array names, which must be contained in the connect sets of the arrays that are redistributed. If it is specified without a *notransfer-list*, it applies to all arrays of the corresponding connect sets.

2.4.5 Allocatable Arrays

Dynamic memory allocation can be achieved in Fortran 90 by means of *ALLOCATABLE* arrays or *POINTERS*. This allows writing programs which are dynamically scalable with respect to the size of data structures. In Vienna Fortran 90 allocatable arrays may be distributed in the same way as ordinary arrays.

Let us first consider some properties of statically distributed allocatable arrays. The distribution type of a **statically distributed allocatable array** is defined within its declaration as described in Section 2.4.1.1. At the time the array is allocated, its index domain is defined and it is associated with a distribution. Since a statically distributed allocatable array may change its index domain through the execution of an *ALLOCATE* statement, its distribution may change too. However, the distribution of such an array is invariant with respect to a particular allocation instance. Furthermore, the change of distribution caused by a modification of the index domain does not alter the distribution type of an allocatable array. As a consequence, the distribution type of a statically distributed allocatable array is fixed within the scope of declaration.

A **dynamically distributed allocatable array** has to be declared with the *DYNAMIC* attribute. The distribution of such an array may be changed by executing a *DISTRIBUTE* or *ALIGN* statement, an *ALLOCATE* statement or as the result of a procedure call. Note that as in the case of statically distributed allocatable arrays the distribution type is not changed as a result of executing an *ALLOCATE* statement. In order to enable the specification of a distribution (or alignment) for dynamically distributed allocatable arrays at the time of allocation, the syntax of the Fortran 90 *ALLOCATE* statement has been slightly modified.[20]

The execution of an *ALLOCATE* statement for a statically distributed array always associates the array with a distribution. Note that an error condition may occur if some constraints imposed by the distribution specification are not fulfilled at the time of allocation. In this case, the *STAT* variable (if specified) will return a value not equal zero.

If a distribution is specified within the declaration of an allocatable array, then the values of all variables referenced in the distribution or alignment specification are evaluated at the time of declaration and are retained until the execution of the allocate statement. This is the case for M in Example 2.4.22. Note that an alignment relationship between two statically distributed allocatable arrays may be destroyed if the source array is reallocated.

[20] Such arrays must not have the *CONNECT* attribute.

Example 2.4.22: Allocation of distributed arrays

```
REAL, ALLOCATABLE, DISTRIBUTED(BLOCK(M))          :: A(:)
REAL, ALLOCATABLE, DYNAMIC, DISTRIBUTED(=A)       :: B(:)
REAL, ALLOCATABLE, DYNAMIC                        :: C(:), D(:,:), E(:)

ALLOCATE(A(N), B(N), C(N)(CYCLIC), D(N,N)((I,J) WITH A(I)), E(N) )
```

In this example array A is a statically distributed allocatable array and all others are dynamically distributed allocatable arrays. B is specified with an initial distribution. After the execution of the ALLOCATE statement array A is associated with a distribution since it is statically distributed. Array B is associated with the initial distribution specified in its declaration, yielding BLOCK(M)*, and arrays C and D are associated with the distribution specified within the ALLOCATE statement. For array E the execution of the ALLOCATE statement does not result in an association of E with a distribution. As a consequence, it must not be accessed unless it has been given a distribution.*

■

The execution of an allocate statement for a distributed array A results in the association of A with a distribution if

- it is a statically distributed array, or
- it is a dynamically distributed array for which an initial distribution[21] has been specified within its declaration, or
- for which a distribution or alignment specification is provided in the allocate statement.

A dynamically distributed array that is declared with an initial distribution is associated with this initial distribution only if a distribution or alignment specification is not specified in the allocate statement. If a distribution or alignment is specified in the allocate statement, it overrides the initial distribution. Note that a dynamically distributed allocatable array must not be accessed unless it has been allocated and distributed.

Furthermore, if a dynamically allocated primary array is associated with a new distribution through the execution of an allocate statement, all secondary arrays in its connect set may have to be redistributed too. This may happen in those cases where the connection is established by means of an alignment relationship. This is demonstrated in Example 2.4.23.

[21] Initial distributions are evaluated at the time of the declaration.

Example 2.4.23: Allocation of connected arrays

```
REAL, ALLOCATABLE, DYNAMIC, DISTRIBUTED(CYCLIC(M))    :: A(:)
REAL, ALLOCATABLE, DYNAMIC, CONNECT(B(I,:) WITH A(I)) :: B(:,:)

ALLOCATE(A(N)); ALLOCATE(B(N,M))   ! s
....
DEALLOCATE(A); ALLOCATE(A(2*N))    ! s'
```

After the execution of statement s an alignment relationship α_A^B between array B and array A has been established

Since such a relationship is guaranteed to stay invariant within the scope of declaration, an implicit redistribution for array B might be necessary after execution of the allocate statement in s'. ∎

2.4.6 Derived Types

In Vienna Fortran 90 an object of derived type may be distributed in the same way as objects of intrinsic types as long as no component at any level of component selection has a distribution associated with it. In the following we show how components of a derived type can be associated with a distribution.

2.4.6.1 Distribution of Components of Derived Types

In Vienna Fortran 90 a distribution for a component of a derived type, say C, may be specified at the point of the derived type definition

- if C is of intrinsic type, or
- if C is of derived type, say T, and for no component of T at any level of component selection a distribution has been specified.

Definition 2.4.2: Distributed Derived Types

A derived type is called distributed derived type iff there exists a component at any level of component selection for which a distribution has been specified. ■

For an object of distributed derived type a distribution must not be specified. This avoids situations as shown in Example 2.4.24.

Example 2.4.24: Distribution of objects of derived type

```
PROCESSORS R(100)
TYPE VECTOR
     REAL, DIMENSION(N), DISTRIBUTED(BLOCK) :: V
END TYPE VECTOR

TYPE(VECTOR)                     :: X(100)  ! (a)
TYPE(VECTOR), DISTRIBUTED(BLOCK) :: Y(100)  ! (b) illegal !!
```

In example (a) array X is of distributed derived type $VECTOR$ which has a component V for which a distribution has been specified. As a consequence, each data object $X\%V(J)$ will be allocated on processor $R(J)$.

Example (b) is illegal since a distribution is specified for Y which, however, is an object of distributed derived type. This results in the following conflict:

- *The distribution for array Y specifies that element $Y(I)$ is mapped to processor $R(I)$, $1 \leq I \leq 100$. As a consequence, all $Y(I)\%V(J)$, $1 \leq J \leq 100$, are owned by processor $R(I)$.*
- *The distribution specification within the derived type definition for the component V, states however that element $X(I)\%V(J)$ is mapped to processor $R(J)$, $1 \leq I, J \leq 100$.*

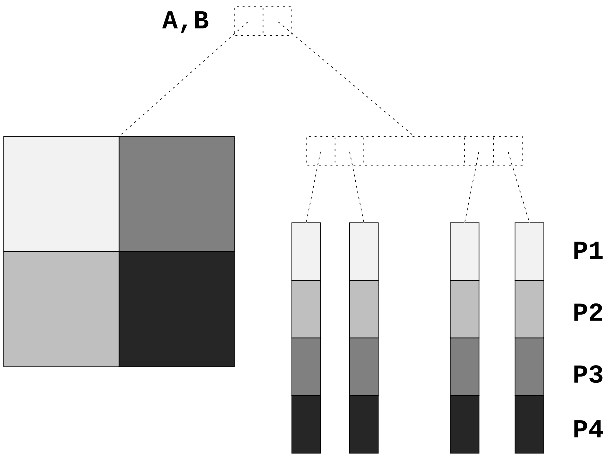
■

Statically Distributed Derived Types

If in a distributed derived type *all* components, at any level of component selection, for which a distribution has been specified are statically distributed, then it is called a **statically distributed derived type**. A statically distributed derived type is constructed by specifying the *DISTRIBUTED* or *ALIGNED* attribute for at least one component, at the time of the derived type definition.

Corresponding subobjects of different objects of a statically distributed derived type have the same distribution type associated with them at any point in the program. Example 2.4.25 illustrates how objects of derived type with distributed components at different levels of component selection can be constructed.

Example 2.4.25: Distribution of components of derived type




```
PROCESSORS R(4),R2(2,2)

TYPE T
   REAL, DISTRIBUTED(BLOCK,BLOCK) TO R2 :: MAT(N,N)
   TYPE(VECTOR)                         :: VEC(M)
END TYPE T

TYPE(T)      :: A,B
TYPE(VECTOR) :: X
```

In this example A, B and X are objects of distributed derived type. (The definition of type VECTOR is taken from example 2.4.24.) Note that forall $I, J, 1 \leq I, J \leq M$, the arrays $A\%VEC(I)$, $B\%VEC(J)$, and X always have identical distributions. The layout of A and B is illustrated in the above figure. ■

It may however often be necessary that different objects of the same derived type are statically distributed to different subsets of processors. To achieve this we have modified the derived type definition of Fortran 90 such that derived types may be parameterized with a processor array.

In Vienna Fortran 90 a derived type may be parameterized by specifying a *dummy processor array* within the derived type definition statement. The dummy processor array is associated with an actual processor array section by means of the *LOCATION attribute*. This attribute has to be specified within the type declaration statement, or if it is specified for a component of a derived type, within the component definition statement. It has the effect that the specified processor array section is *reserved* for all objects of that derived type. All subobjects of such an object are distributed with respect to the specified processor array section.

Assume that a derived type T is parameterized with a processor array Q as shown in example 2.4.26. The dummy processor array Q is associated with an actual processor array section by means of the *LOCATION* attribute. As a consequence, Q becomes the only processor array accessible within the derived type definition.[22] Note that the *LOCATION* attribute may only be specified within the declaration of objects of statically distributed derived type. If it is omitted, the primary processor array is associated with Q by default.

Example 2.4.26: The *LOCATION* attribute

```
PROCESSORS R(1:100)

TYPE (T), PROCESSORS Q(:)
   REAL, DISTRIBUTED(BLOCK) TO Q(1:SIZE(Q)/2) :: X(M)
   TYPE(T2), LOCATION(Q(SIZE(Q)/2+1:))        :: Y
END TYPE

TYPE(T), LOCATION(R(1:50))   :: A
TYPE(T), LOCATION(R(51:100)) :: B
TYPE(T)                      :: C
```

Array $A\%X$ is distributed to the first 25 processors of R whereas $B\%X$ is distributed to processors $R(51:75)$. Since C is declared without a LOCATION attribute, $C\%X$ is distributed over the whole processor array R. Furthermore, components X and Y are distributed to disjoint processor subsets. ■

If a processor declaration occurs within a derived type definition statement, it must be of assumed shape. Furthermore, the rank of the actual processor array (section) and the dummy processor array must match. An entity which has the *LOCATION* attribute must not have the *DISTRIBUTED*, *ALIGNED*, or *DYNAMIC* attribute.

In the following we discuss the impact of objects of derived type which have dynamically distributed components.

[22] Different views of the dummy processor array may be provided by using the *RESHAPE* function.

Dynamically Distributed Derived Types

If in a distributed derived type *any* component, at any level of component selection, is dynamically distributed then it is called a **dynamically distributed derived type**.

Structure components which have the *DYNAMIC* attribute may be associated with a new distribution by executing a *DISTRIBUTE* or *ALIGN* statement or as the result of a procedure call. Furthermore, an initial distribution may be specified in the same way as discussed in Section 2.4.1.1. Given an array of dynamically distributed derived type, corresponding components of different array elements may be associated with different distributions. This is explained in Example 2.4.27.

Example 2.4.27: Derived types with dynamically distributed components

```
TYPE T
   REAL, DYNAMIC :: D(M,M)
   ...
END TYPE T

TYPE(T) :: A(N)

DO I=1,N                           ! s1
   DISTRIBUTE(A(I)%D(BLOCK,:))
END DO

DISTRIBUTE(A%D(BLOCK))             ! s2
```

Since D is dynamically distributed, each array $A(I)\%D$, $1 \leq I \leq N$, may be associated with a different distribution during run-time. The do-loop (s1) has exactly the same effect as the second distribute statement (s2); all arrays $A(I)\%D$ are associated with the same distribution. Note that the data-ref *in statement (s2) would be illegal in Fortran 90 but is allowed in Vienna Fortran 90.* ∎

In order to associate distributed components of an array with identical distributions in parallel, without the need of specifying a do loop (see Example 2.4.27), we have relaxed the following constraint of Fortran 90 ([72], 6.1.2) on *data-ref*s:

> In a *data-ref* there must not be more than one *part-ref* with no-zero rank. A *part-name* to the right of a *part-ref* with non-zero rank must not have the *POINTER* attribute.

In Vienna Fortran 90 this constraint does not apply to the objects of a *DISTRIBUTE* or *ALIGN* statement. If a structure component is the object of a distribute or align statement, more than one *part-ref* may have non-zero rank. However, all other references to structure components must obey the above constraint.

Furthermore, the distribution of a component of a derived type may be defined implicitly by referring to the distribution of a structure component on a deeper level of the same derived type definition, as shown in Example 2.4.28. In order to alleviate in such cases the specification of an implicit distribution, a *subobject designator* that appears in an alignment specification or in a distribution extraction may have more than one *part-ref* with non-zero rank as long as the right most *part-ref* does not have the *DYNAMIC* attribute.

Example 2.4.28: Implicit distribution of components

```
TYPE T1
   REAL, DISTRIBUTED(BLOCK) :: SD(N)
   REAL, DYNAMIC            :: DD(N)
END TYPE T1

TYPE T2
   TYPE (T1)                         :: S, A(N)
   REAL, DISTRIBUTED(=S%SD)          :: C1(N)
   REAL, DISTRIBUTED(=A%SD)          :: C2(N)
   REAL, DYNAMIC, CONNECT(=A%DD) :: C3(N)     ! not allowed
END TYPE T2
```

In this example the components C1, C2, and C3 are implicitly distributed by referring to the distribution of subcomponents of S and A. Note that the distribution extraction in the DISTRIBUTED attribute of array C2 designates a data-ref with two part-refs of non-zero rank but is valid since the rightmost part-ref designates a statically distributed subobject. On the other hand the distribution extraction in the CONNECT attribute of array C3 is illegal since there is more than one part-ref with non-zero rank and the right-most part-ref designates a subobject that has the DYNAMIC attribute. ■

Objects of derived type that have *POINTER* components are discussed in the next section.

2.4.7 Pointers

One important feature of Fortran 90, which for example is very useful for multigrid applications, is that structure components can be dynamically allocated arrays. Since in Fortran 90 the *ALLOCATABLE* attribute must not be specified for a component of a derived type, the *POINTER* attribute has to be used to declare a structure component that is a dynamically allocated array. Furthermore, the use of pointers makes it possible to hide memory allocation inside of procedures.

A much more powerful feature of Fortran 90 pointers is to use them as aliases for arrays or array sections. An array pointer of rank n can be associated with any array or array section[23] of rank n by means of the pointer assignment. Every object occurring on the right hand side of a pointer assignment must be declared with the *TARGET* attribute or must also be a pointer.

In analogy to ordinary arrays, array pointers may be associated with a distribution either statically or dynamically. However, Vienna Fortran 90 allows using statically distributed pointers only in a restricted form. This is discussed in more detail in the following.

2.4.7.1 Statically Distributed Array Pointers

Components of derived types that are arrays and for which the *POINTER* attribute has been specified, may be associated with a static distribution in the same way as non-pointer arrays. Statically

[23] Fortran 90 forbids array sections with vector subscripts to be target of a pointer assignment.

distributed objects for which the *POINTER* attribute has been specified are called **statically distributed pointers (SDPs)**.

An SDP must not be the left hand side of a pointer assignment statement, since this could cause the SDP to become associated with a different distribution. A distribution for an SDP is specified in the same way as for statically distributed arrays by means of the *DISTRIBUTED* attribute or *ALIGNED* attribute. Note that a statically distributed array pointer must not be accessed unless it has been allocated.

SDPs have been included in Vienna Fortran 90 for two reasons. First, they enable that components of derived types that are dynamically allocated arrays may be distributed and second, dynamic memory allocation may be hidden inside of a procedure. In Example 2.4.29 the declaration of a multi-grid data structure (i.e an array containing grids of different sizes) is shown.

Example 2.4.29: Array of pointers with identically distributed arrays

```
PROCESSORS R2(10,10)

TYPE GRID
     REAL, DISTRIBUTED(BLOCK,BLOCK), POINTER :: U(:,:)
     REAL, DISTRIBUTED(=U), POINTER          :: F(:,:)
END TYPE GRID

TYPE (GRID) :: MG(10)

ALLOCATE(MG(I)%U(N,N), MG(I)%F(N,N) )   ! allocate & distribute i-th grid
```

In this example the derived type GRID consists of two statically distributed array pointers. This kind of data type is useful in multigrid applications where the sizes of the grids U and F are determined at runtime, depending on the grid level. ∎

2.4.7.2 Dynamically Distributed Array Pointers

An object that has both the *POINTER* and *DYNAMIC* attribute is called a **dynamically distributed pointer** (DDP). A dynamically distributed pointer may be the alias of more than one distributed array (section).

In contrast to SDPs, DDPs may be associated with sections of distributed arrays by means of the pointer assignment statement. Therefore, the distribution associated with a DDP may be changed at run time through the execution of a pointer assignment statement, an *ALLOCATE* statement, a *DISTRIBUTE* or *ALIGN* statement, or as the result of a procedure call.

Using DDPs, Example 2.4.29 can be modified in such a way that depending on the size of the grids on different levels they are distributed to different processor subsets.

Example 2.4.30: Array of pointers to individually distributed arrays

```
PARAMETER (NP1 = 10)
PROCESSORS :: R(NP1,NP1)

TYPE GRID
     REAL, POINTER, DYNAMIC              :: U(:,:)
     REAL, POINTER, DYNAMIC, CONNECT(=U) :: F(:,:)
END TYPE GRID

TYPE (GRID), ALLOCATABLE :: MG(:)

ALLOCATE(MG(MAXG))
N = 2**(MAXG) + 1

DO I=1, MAXG
   M = MAXG-I+1
   ALLOCATE( MG(I)%U(N,N)(BLOCK,BLOCK) TO R2(:M,:M), MG(I)%F(N,N) )
   N = (N + 1) / 2
END DO
```

In this example each element of array MG consists of a pointer to a dynamically distributed array. What is important here is that each array MG(I)%U is distributed depending on its size to a different subset of the available processors

Assuming MAXG = 10 we have the following grids:

finest grid	*(1025 x 1025)*	*on*	*R(1:10,1:10)*
	(513 x 513)	*on*	*R(1:9,1:9)*
...	...		...
coarsest grid	*(3 x 3)*	*on*	*R(1,1)*

■

Pointer Assignment

In Vienna Fortran 90 only pointers that have the *DYNAMIC* attribute may be associated with other distributed objects by means of a pointer assignment statement. A dynamically distributed pointer may be associated with either a statically or dynamically distributed array, as shown in the previous example. If a DDP appears on the left hand side of a pointer assignment, it inherits the distribution and the association status of the target or of the pointer on the right hand side. A pointer that does not have the *DYNAMIC* attribute must not be associated with a distributed array in this way.

If a DDP is associated with a section of a distributed array, then it usually will be associated with a new distribution. The mechanisms to determine the distribution of a pointer to an array section are the same as those encountered when an array section is passed to a procedure. A DDP that is currently associated with a statically distributed object must not be associated with a new distribution. In the following we give some examples to illustrate the effects of pointer assignments in the context of distributed arrays.

Example 2.4.31: Pointers to sections of distributed arrays

```
PARAMETER (NP=4, N=NP*100)
PROCESSORS R(NP1*NP1), R2(NP1,NP1)

REAL, DISTRIBUTED(BLOCK,BLOCK) TO R2, TARGET :: A(N,N)
REAL, DYNAMIC, POINTER                       :: COL(:), ROW(:)

DO I=1,10
   ROW=>A(I,:) ! ROW has distribution "(BLOCK) TO R2(J,:)"
   COL=>A(:,I) ! COL has distribution "(BLOCK) TO R2(:,J)"
   DISTRIBUTE(ROW(somehow))  ! illegal
   ...
END DO
```

In this example COL and ROW are declared as dynamically distributed pointers and are associated with sections of the statically distributed array A within a do loop. Inside the do loop array ROW, being an alias for A(I,:), has a BLOCK distribution with respect to processors R2(J,:), with $J = \left\lceil \frac{I}{NP} \right\rceil$. *Note that the distribute statement is illegal since ROW is currently associated with a statically distributed array.*

```
PROCESSORS R(10)
INTEGER, DYNAMIC, DISTRIBUTED(BLOCK), TARGET :: B(100)
INTEGER, DYNAMIC, POINTER                    :: P(:)

P => B(::3)  ! P has distribution S_BLOCK((/(4,3,3,I=1,3),4/))
```

In this example P is associated with a section of a dynamically distributed array. This yields a general block distribution for P (see Section 2.4.1.1). ■

Redistribution of Pointers

A dynamically distributed pointer may be associated with either a statically or dynamically distributed array. A DDP may be the object of a distribute statement if and only if it is currently associated with a dynamically distributed object. If a pointer is the object of a distribute statement, then the target is implicitly redistributed. The same situation arises if a dynamically distributed pointer it is associated with a new distribution as the result of a procedure call. A pointer that is currently associated with a statically distributed object must not be associated with a new distribution by execution of a *DISTRIBUTE* statement or as the result of a procedure call. Such an illegal redistribution is shown in Example 2.4.31.

Furthermore, if a DDP, say P, is associated with a distributed array A which subsequently gets redistributed by other means than by redistributing P, then P must not be referenced unless it is again associated with a distributed array.[24]

[24] In analogy to dangling pointers that have been associated with an array prior to the deallocation of this array.

Example 2.4.32: Redistribution of pointers

```
REAL, TARGET, DYNAMIC, DISTRIBUTED(BLOCK) :: A(N), B(N)
REAL, POINTER, DYNAMIC                    :: P(:), Q(:)

P => A
Q => B(1:100:2)
...
DISTRIBUTE(A(CYCLIC))  ! (s1)
...
DISTRIBUTE(Q(CYCLIC))  ! (s2)
```

In this example A gets redistributed (s1) after the pointer P has been associated with it. As a consequence, P must not be accessed unless it has been associated with a distributed array again.

Pointer Q is associated with a section of array B. The execution of the distribute statement (s2) results in an implicit redistribution of the target array B. Note that the distribution of B after the execution of the distribute statement can no longer be described by means of an intrinsic distribution functions, since the even elements are BLOCK distributed and the odd elements have a CYCLIC distribution. However, the distribution can easily be described by means of a mapping procedure. ■

2.4.8 Distribution Type Definitions

A distribution type (or a class of them) can be bound to a name by means of the distribution type definition. A distribution type definition may have arguments in order to allow for parameterization. Since a derived type definition is a declaration construct, it may appear only in the specification part of a program unit. In accordance with a derived type definition, a distribution type definition is regarded as a separate scoping unit.

Example 2.4.33: Distribution type definition

```
DTYPE DT4                              DTYPE BICYCLE(M,N)
   INTEGER, PARAMETER :: K = 10           INTEGER M,N
   DT4 = (BLOCK,:,CYCLIC(K),:)            BICYCLE = (CYCLIC(M),CYCLIC(N))
END DTYPE DT4                          END DTYPE

DTYPE DT3
   INTEGER, PARAMETER :: BB(10) = (/ 10,15,20,27,33,40,45,78,90,94 /)
   DT3 = (B_BLOCK(BB),:,:)
END DTYPE
```

The distribution type names DT4, DT3, and BICYCLE can be used in a declaration of a distributed array, as shown below.

```
REAL, DISTRIBUTED(DT4)           :: A(N,M,N,N)
REAL, DISTRIBUTED(DT3)           :: B(M,N,N)
REAL, DISTRIBUTED(BICYCLE(5,10)) :: C(N,N)
```

■

If a distribution type definition does not contain any dummy arguments, the parentheses in the distribution type definition can be omitted. Note that there is no implicit typing for dummy arguments of distribution type definitions. Therefore, each dummy argument must be declared explicitly. Besides the dummy arguments only scalar- or array-constants of type integer may be declared.

A reference to a defined distribution type may appear in any place where a reference to an intrinsic distribution function, like for example *BLOCK*, may appear.

2.4.9 Defaults

If the *DISTRIBUTED* attribute is specified without the optional processor reference, then the primary processor array is substituted by default[25].

If an object is declared without the *DISTRIBUTED*, *ALIGNED* or *DYNAMIC* attribute, its distribution is determined by the compiler.[26] A *DISTRIBUTED* attribute where the distribution specification consists of colons only specifies that the corresponding array is replicated with respect to the corresponding processor array. If the *DISTRIBUTED* attribute is specified for scalars, a distribution specification must not be present.

Example 2.4.34: Distribution of scalars

```
REAL, DISTRIBUTED TO R(::2)  :: S
```

This declaration designates every other processor of R as the owner of S. ■

2.4.10 Undefined Distributions

An array A has an **undefined distribution** iff at least one of the following conditions are met:

- A is allocatable and has not yet been allocated
- A is a pointer and has not yet been allocated or associated with a target
- A is dynamically distributed and has not yet been associated with a distribution

An array with an undefined distribution must not be accessed. Note that Vienna Fortran 90 allows to hide the distribution and allocation (or pointer association) of an array inside a procedure. This is discussed in the next section.

[25] This is not the case for dummy arrays with assumed distribution.

[26] The compiler may for example replicate the corresponding object.

2.5 Procedures

In Vienna Fortran 90 a procedure may either be internal or external.[27] Note that internal procedures must not be nested and their names are not globally available. They also must not be passed as arguments to procedures. Interface blocks may (or under certain restrictions must) be specified for external procedures. Fortran 90 features like optional or keyword arguments, or generic features are not supported by this version of Vienna Fortran 90.

The distribution of local entities and dummy arguments in procedures can be specified in precisely the same way as for entities in the main program. This allows viewing objects within a procedure independently of the distribution of the corresponding actual arguments. Every reference to a formal array inside a procedure is interpreted according to the local declaration. Local arrays of procedures may either be distributed explicitly or they may be aligned or connected to dummy arrays.

In addition, a dummy argument may inherit its distribution from the actual argument. This enables the user to write procedures which will accept actual arguments with a variety of distributions: within the procedure, the distribution of the actual argument may be used, or a specific distribution may be enforced, as required. Distribution of dummy arrays is discussed in more detail in Section 2.5.3.

In order to explicitly control the mechanisms of parameter passing and to override defaults, Vienna Fortran provides additional attributes, which may be specified either within the call statment or within the declaration of dummy arrays, as explained in the next sections.

2.5.1 Interfaces

A procedure interface is the collection of information that is needed by a compiler to generate a correct call to the procedure. The interface of internal procedures is always known (i.e. explicit) to the compiler. Vienna Fortran 90 requires that the interface of a procedure that has distributed dummy arrays is explicit. Furthermore, procedure local processor array declarations must be specified in an interface.

The interface of an external procedure must be made explicit by providing an interface block if any of the following conditions is met:

- The procedure has a dummy argument
  - that is an assumed shape array, or
  - that has the *POINTER* or the *TARGET* attribute, or
  - that has the *DISTRIBUTED*, *ALIGNED*, *DYNAMIC*, or *LOCATION* attribute.
- The result is array valued or a *POINTER*.
- It contains a processor declaration.

[27] This version of Vienna Fortran 90 does not include module procedures.

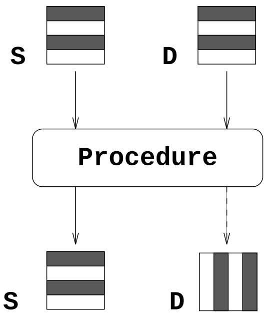
2.5.2 Transfer of Distributed Arguments

While the principal mechanism of parameter passing is call-by-reference, there are some situations (e.g. when a section of an array is passed to a subroutine and redistributed) in which the compiler may have to copy arguments to ensure that the correct distribution is preserved upon exit from the procedure. Note that the Fortran 90 Standard explicitly permits this.

The distribution of statically distributed arrays never changes as the result of a procedure call. Dynamically distributed actual arrays may, however, have a modified distribution after a procedure call if the corresponding dummy array has been redistributed inside the procedure. Whereas this allows to define or modify the distribution of dynamically distributed arrays inside procedures, unrestricted use of this feature may seriously complicate the task of the compiler to determine which distributions may reach a particular array reference and thus requires sophisticated interprocedural analysis. Restoration of the original distribution can always be enforced by specifying the *RESTORE attribute*, either in the formal parameter declaration or with the actual argument at the point of the call.

Example 2.5.1: Distribution of actual arrays

```
DISTRIBUTED(BLOCK,:) :: S,D
DYNAMIC              :: D
...
CALL PROCEDURE(S,D)
...
CONTAINS
  SUBROUTINE PROCEDURE(X,Y)
    ...
    DISTRIBUTE(D(:,BLOCK)) ! illegal
```



A statically distributed array (S) never changes its distribution as the result of a procedure call. However, the distribution of a dynamically distributed array (D) may change through a procedure call. The distribute statement is illegal since D is associated with a dummy argument of the subroutine. ■

Note that an internal procedure also may change the distribution of all dynamically distributed arrays that are accessible by host association or by use association as long as they are not associated with a dummy argument. The restrictions of Fortran 90 on entities associated with dummy arguments have been extended in order to cover the distribution of an entity. They are included here for sake of consistency.

Restrictions on Entities Associated With Dummy Arguments [72](12.5.2.9):

- While an entity is associated with a dummy argument no action may be taken that affects the value, availability, or *distribution* of the entity or any part of it, except through the dummy argument.

- If any part of the entity is defined through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument.

2.5.3 Distribution of Dummy Arrays

Distributions for dummy arguments may be specified using the same mechanisms as provided for actual arrays. In this case, we speak of an explicitly distributed dummy array. An explicit distribution of a dummy array is always enforced on procedure entry. Furthermore, a dummy array may inherit the distribution from the actual array. In accordance with the terminology of Fortran 90, this is referred to as an *assumed-distribution array.*

2.5.3.1 Dummy Arrays with Explicit Distribution

If a dummy argument is statically distributed, the specified distribution is enforced on procedure entry - this will cause an implicit redistribution if the actual argument does not have a matching distribution. If the dummy argument is dynamically distributed with an initial distribution, then the same holds. The distribution of a dynamically distributed dummy array that is declared without an initial distribution is defined by the distribution of the actual argument (which may be undefined). For dummy arrays with the *DYNAMIC* attribute a distribution range may be specified in the same way as discussed in Section 2.4.4.3.

Example 2.5.2: Explicitly distributed dummy arrays

```
SUBROUTINE SUB(X,Y,Z)
REAL, DISTRIBUTED(BLOCK)           :: X(:)
REAL, DYNAMIC, DISTRIBUTED(CYCLIC) :: Y(:)
REAL, DYNAMIC, ALIGNED(WITH X)     :: Z(:)
```

In this example X is a statically distributed array, Y is a dynamically distributed array with an initial distribution specified, and Z is a dynamically distributed array which is initially aligned with X.

■

In the following we briefly discuss the features provided by Vienna Fortran 90 to control the mechanisms of parameter passing and to override defaults.

If a redistribution is required for an explicitly distributed dummy array on procedure entry, then the redistribution can be performed without the need of exchanging data between the processors if the dummy array has the *INTENT(OUT)* attribute. In this case, only the access function is changed but no physical movement of data takes place. Note that an array that has the *INTENT(OUT)* attribute becomes undefined on procedure entry and therefore must not be read before it is defined.

The *RESTORE* attribute can be used to enforce that the distribution of a dynamically distributed array is not modified through a procedure call. It may be specified either for actual arrays[28]

[28]The actual array must be dynamically distributed and must have a defined distribution.

at the point of a procedure call, or within the declaration of dummy arrays. Statically distributed arrays always have the restore property. It is an error to associate an array with an undefined distribution with a dummy array that has the *RESTORE* attribute.

Example 2.5.3: The *RESTORE* attribute

```
REAL, DYNAMIC, DISTRIBUTED(BLOCK) :: A(100)

CALL KEEPIT(A::RESTORE)
...
   SUBROUTINE KEEPIT(DDA)
      REAL, DYNAMIC, DISTRIBUTED(CYCLIC), RESTORE :: DDA
      ...
```

In this example the RESTORE attribute is specified for the actual argument at the point of the call and also in the declaration of the dummy argument. Since the dummy array DDA has an explicit distribution specified that does not match A's distribution, an implicit redistribution has to be performed on procedure entry. On procedure exit a second implicit redistribution might take place because the BLOCK distribution of A has to be restored.

■

2.5.3.2 Dummy Arrays with Assumed Distribution

According to assumed shape arrays of Fortran 90, dummy arrays may have an assumed distribution. Dummy arrays with an assumed distribution inherit the distribution of the actual array and no implicit redistribution takes place on procedure entry. Assumed distributions are specified by using the "*" for the distribution specification. An array with an assumed distribution is a statically distributed array and therefore must not be the object of a distribute statement even if the corresponding actual argument is dynamically distributed. It is illegal to associate an array with an undefined distribution with an assumed distribution array.

Example 2.5.4: Dummy arrays with assumed distribution

```
SUBROUTINE FLEXIBLE(X,Y)
   REAL, DISTRIBUTED(*)                              :: X(:,:)
   REAL, DISTRIBUTED(*), RANGE((BLOCK,:),(CYCLIC(*),:)) :: Y(N,N)
```

Both dummy arrays are specified with an assumed distribution. Array X may be associated with an actual array with an arbitrary distribution, whereas array Y may be associated only with actual arrays that are distributed by BLOCK or CYCLIC with any legal block length in the first dimension and undistributed in the second dimension.

■

Assumed distribution arrays may have a different distribution in different incarnations of a procedure. As a consequence, the efficiency of the computation within the procedure may depend heavily on the actual distribution of the argument. Interprocedural distribution analysis [49, 51] will, however, often reveal the distributions reaching a procedure, and enable the compiler to optimize the target code.

In order to supply the compiler with additional information, the set of all possible argument distributions can be specified via the *RANGE* attribute. If the *RANGE* attribute is specified, an associated actual array must have a distribution that is contained in the range. A distribution range that contains exactly one distribution specification constitutes an assertion to the compiler that the actual array will always have the distribution described in the range. The use of inquiry functions or the *DCASE* construct as explained in Section 2.7 facilitates the writing of procedures that are optimal for multiple incoming distributions.

2.5.3.3 Passing Array Sections

If an array section is passed to a procedure, then the argument transfer will be call by reference iff the corresponding actual array is an assumed shape array and the distribution of the actual array and dummy array matches. This is always the case when the dummy array assumes both the shape and the distribution. In all other cases where the shape of the dummy array is specified explicitly or an explicit distribution specification is present a copy-in/copy-out argument transfer mechanism may have to be adopted. Since explicit shape arrays imply contiguous storage and rely on storage association, an actual array section will have to be copied upon procedure entry and copied back on return. Dummy arrays with an explicit distribution that are associated with an array section might enforce a call by copy semantics since the redistribution might not be possible without introducing a temporary array.

If a dummy array is associated with a section of a dynamically distributed array and the dummy is subsequently redistributed, the same effects as discussed in the context of pointer assignment in Section 2.4.1.1 arise.

2.5.3.4 Pointers Associated With Dummy Arguments

Dummy arguments, which have the *POINTER* attribute, must not be specified with the *INTENT* attribute. If a procedure has dummy arguments with the *POINTER* attribute, the interface must be explicit to the caller and the corresponding actual arguments must have the pointer attribute and the types, type parameters, and ranks must agree [72]. Note that a dummy array with the pointer attribute is not regarded as an assumed shape array because its shape can be undefined if the pointer association status of the actual array is undefined.

The major reason for using the *POINTER* attribute for dummy arrays is to hide the allocation (and distribution) of an array inside a procedure.

2.5.3.5 Actions upon Procedure Entry and Exit

In this section we take a closer look at the actions that have to be performed upon procedure entry and procedure exit in case of distributed arguments. Let in the following A denote an actual array and D the dummy array that gets associated with A through a procedure call.

Case 1: Statically distributed dummy arrays with explicit distribution:

1. **Statically distributed actual array:**

   (a) **Procedure Entry:**
   An implicit redistribution has to be performed on procedure entry iff A has a defined distribution which does not match the distribution of D. The redistribution can be performed without physical transfer of data, iff D has the *INTENT(OUT)* attribute.

   If A has an undefined distribution with a target), then the dummy D must have the *POINTER* attribute and no redistribution takes place in this case[29] (Example 2.5.5 s2).

   An error message will be issued if A has an undefined distribution and D does not have the *POINTER attribute* (Example 2.5.5 s1, s3).

   (b) **Procedure Exit:**
   An implicit redistribution has to be performed on procedure exit iff the distribution of D does not match the distribution of A.

2. **Dynamically distributed actual array:**

   (a) **Procedure Entry:**
   An implicit redistribution has to be performed on procedure entry if A has a defined distribution which does not match the distribution of D. The redistribution can be performed without physical transfer of data, iff D has the *INTENT(OUT)* attribute.

   If the distribution of A is undefined and A is a statically allocated array, then D is distributed according to the specified distribution (Example 2.5.6 s1).

   If A has an undefined distribution and is a pointer, then the dummy D must have the *POINTER* attribute and no (re)distribution takes place in this case (Example 2.5.6 s4).

   An error message will be issued if A has an undefined distribution and is dynamically allocated and D does not have the *POINTER attribute* (Example 2.5.6 s3, s5).

   (b) **Procedure Exit:**
   A redistribution has to be performed iff the *RESTORE* attribute has been specified either for the actual array A within the call statement or for the dummy array D. Furthermore, if A is a secondary array (i.e it has the *CONNECT* attribute), its distribution has to be restored upon procedure exit iff the relationship of A's distribution with the distribution of the corresponding primary array has been broken.

   An error occurs on exit if the *RESTORE* attribute has been specified and A had an undefined distribution on entry.

[29] Note that D must not be accessed until it has been associated with a target.

Example 2.5.5: Statically distributed actual arguments with undefined distribution

```
REAL, POINTER, DISTRIBUTED(BLOCK)     :: A1(:)
REAL, ALLOCATABLE, DISTRIBUTED(BLOCK) :: A2(:)

CALL Sub1(A1)   ! s1   ! error
CALL Sub2(A1)   ! s2

CALL Sub1(A2)   ! s3   ! error
CALL Sub2(A2)   ! s4   ! illegal in Fortran 90

SUBROUTINE Sub1(D)                     SUBROUTINE Sub2(D)
REAL, DISTRIBUTED(BLOCK) :: D(:)       REAL, POINTER, DISTRIBUTED(BLOCK) :: D(:)
...                                    ...
```

The call in statement s1 will result in an error since, A1 is not associated with a target (or it has not been allocated). Similarly, the call in s3 will produce an error since A2 has not been allocated. The call in s2 shows how the allocation and/or distribution can be performed inside a procedure. The last call (s4) is not allowed since an non-pointer actual argument is associated with pointer dummy argument. ∎

Example 2.5.6: Dynamically distributed actual arguments with undefined distribution

```
REAL, DYNAMIC                                    :: A0(:)
REAL, POINTER, DYNAMIC, DISTRIBUTED(BLOCK)      :: A1(:)
REAL, ALLOCATABLE, DYNAMIC,  DISTRIBUTED(BLOCK) :: A2(:)

CALL Sub1(A0)   ! s1
CALL Sub2(A0)   ! s2    ! illegal in Fortran 90

CALL Sub1(A1)   ! s3    ! error
CALL Sub2(A1)   ! s4

CALL Sub1(A2)   ! s5    ! error
CALL Sub2(A2)   ! s6    ! illegal in Fortran 90

SUBROUTINE Sub1(D)                     SUBROUTINE Sub2(D)
REAL, DISTRIBUTED(BLOCK) :: D(:)       REAL, POINTER, DISTRIBUTED(BLOCK) :: D(:)
...                                    ...
```

After return from the call in statement s1 A0 will be associated with a BLOCK *distribution. The calls in s2 and s6 are illegal because a pointer dummy must not be associated with a non-pointer argument. Execution of s3 and s5 will produce an error since A1 has not yet been associated or allocated respectively. Note that after the call in s4 the corresponding dummy array must not be accessed until it has been allocated or associated with a target.* ∎

Case 2: Dummy arrays with assumed distribution

> A dummy array with an assumed distribution can be associated with any distribution and therefore neither upon procedure entry nor upon exit a redistribution has to take place. However, if the distribution of the actual array is undefined upon procedure entry and the dummy array does not have the *POINTER* attribute an error message will be issued. Note that if a dummy array that is a pointer gets associated with an undefined distribution then it must not been accessed unless it has been (allocated and) distributed.

Case 3: Dynamically distributed dummy arrays

A dummy array with the *DYNAMIC* attribute for which no initial distribution has been specified takes over the distribution of the actual array. In this case, no redistribution has to occur on procedure entry.

If the dummy array is redistributed within the procedure, a redistribution in order to restore A's distribution may have to be performed upon procedure exit iff

- A is statically distributed, or
- A is a secondary array, or
- the RESTORE attribute has been specified.

If an initial distribution is specified for D, this distribution is enforced upon entry to the procedure. In this case, the mechanisms at the procedure boundaries are exactly the same as for statically distributed dummy arrays. Note however that dynamically distributed dummy arrays may change their distribution within the procedure.

2.5.4 Array Valued Functions

A distribution for the result of an array valued function may be specified in much the same way as for ordinary arrays - either directly or implicitly by referring to the distribution of a dummy array. Unless the bounds of an array valued function are not constant its shape and thus its distribution my be different in each incarnation. The same holds if the distribution of the function result is specified implicitly by referring to the distribution of a dummy argument. The bounds of the function result are evaluated upon function entry provided that the *POINTER* attribute has not been specified. Note that the interface of an array valued function must be explicit.

2.6 Forall Loops

VF90 provides an explicitly parallel forall loop enabling the user to assert that the iterations of a loop are independent and can be executed in parallel. No value written in one iteration of such a loop may be read or written in another iteration, unless special reduction statements are used which permit global operations to be performed. The result of such an operation is not available within the loop. An intrinsic reduction function and intrinsic reduction operations are provided

(*ADD, MULT, MAX* and *MIN*). The user may define further such operations. Moreover, the user may explicitly specify how the iterations of a forall loop should be distributed to the available set of processors.

The body of a forall loop may contain any legal Fortran statement and in addition calls to *local procedures* (see Section 2.9). Note that there is an implicit synchronization at the beginning and at the end of a forall loop.

Forall loops are especially important in the context of irregular applications where large data arrays are accessed indirectly. In such cases the compiler usually cannot determine if the corresponding loops contain loop carried dependencies and thus will fail to parallelize such loops.

Forall loops may be regarded as a generalization of the Fortran 90 array assignment since they allow to express more general array sections than achievable with triplet notation. The only difference of Vienna Fortran 90 forall loops to those defined in [139] is that in addition to the *on clause* the work distribution may be specified in much the same way as the distribution of arrays. Therefore, we discuss here only those features that are relevant for specifying the work distribution. The complete semantics of the Vienna Fortran forall loop is defined in [139].

The distribution of the iterations of a forall loop may be specified explicitly either with an *on-clause* or by means of a distribution specification. If the work distribution is not explicitly specified, it will be determined by the compiler. Within the on-clause a processor reference or the intrinsic function *OWNER*, which takes an array element as argument and returns the processor index (with respect to \$P) of the processor that owns[30] this array element, may be used.

Furthermore, if the loop iteration space of is rectilinear the distribution of the loop iterations may be specified by means of a distribution specification in much the same way as for the distribution of data arrays. Such work distribution specifications may be either composite or general, and may contain references to intrinsic distribution functions or user defined mapping functions.

Let us assume a forall statement of the following form:

$$\texttt{FORALL(}I_1 = L_1, U_1, S_1, \ldots, I_n = L_n, U_n, S_n\texttt{) DISTRIBUTED(}\mathcal{M}_1, \ldots, \mathcal{M}_n\texttt{) TO R}$$

where $\mathcal{M}_i$ are references to mapping functions and R denotes an m-dimensional processor array, $m \geq n$. The effect of the above work distribution specification is as follows:

Let $\mathbf{I}^{\mathcal{L}} = [L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n]$ denote an index domain that represents the loop iteration space. The iterations $\mathbf{i} \in \mathbf{I}^{\mathcal{L}}$ are distributed to a non-empty set of processors with indices $\mathbf{j} \in \mathbf{J}^R$ as defined by the semantics of composite distributions of Section 2.4.1.1. The implicit arguments of the mapping functions $\mathcal{M}_i$ are associated with $[L_i : U_i : S_i]$ and $\mathbf{J}_i^R$ respectively.

[30] If an array element is owned by more than one processor, an implementation specific processor index will be returned.

2.7 Distribution Inquiries

In order to enable the user to write code which is executed only when an array has a certain distribution type, the control constructs have been extended. This allows to formulate algorithms depending on the actual distribution types of data objects. In addition, the compiler may produce more efficient code since the use of such constructs makes the actual distribution of an array explicit. For this purpose Vienna Fortran provides the *DCASE* construct which is an extension of the Fortran 90 *CASE* construct. The use of the *DCASE* construct is especially important in the context of procedures that have dummy arrays with assumed distribution. It allows the user to write code which is executed only when a dummy array that inherits the distribution of the actual argument has a specific given distribution.

Syntactically, the *CASE* syntax has been modified in such a way that instead of the *case expression* a list of arrays may be specified. In the *dcase selectors* a list of distribution specifications has to be supplied. The distribution type of each of these arrays is compared with the corresponding distribution specification in the list following the keyword CASE. The distribution specification in the *dcase-selector* is identical to a distribution specification with the difference that the '*' may be used as a wild-card symbol with the same meaning as explained for the *RANGE* attribute in Section 2.4.4.3.

As with *CASE* statements, at most one *CASE* selector may be matched. The array names in the *SELECT DCASE* statement are matched with the distribution specification in the *CASE* selectors either by position or by name-tag. The *DEFAULT* selector matches if none of the previous selectors have been matched. If no match occurs, control passes to the next executable statement following the *END SELECT* statement. A *DCASE* statement is shown in example 2.7.1. In addition, the logical intrinsic functions *IDT* and *IDTA* can be used to compare the distribution type of arrays. For the precise definition of these constructs the reader is referred to [139].

Example 2.7.1: The *DCASE* construct

```
REAL, DYNAMIC :: A(N), C(M)
 ...
SELECT DCASE ( A, C )
CASE (BLOCK), (BLOCK)
 ...                    ! efficient code for block distributions
CASE (CYCLIC), *
 ...                    ! efficient code for cyclic A and any C
CASE DEFAULT
 ...                    ! general, less efficient code
END SELECT
```

In this example, the distributions of A and C are first checked to see whether they have a BLOCK *distribution. If so, the corresponding CASE is executed and control passes to the end of the* DCASE. *Otherwise, the distribution of A is checked to see whether it is* CYCLIC*: if so, the next CASE will be executed no matter what distribution C has. If neither is satisfied, the DEFAULT is executed.* ■

2.8 Modules

Fortran 90 has a number of language constructs to support modularization. Modules provide a further means of sharing data and make the use of *COMMON* blocks and *BLOCK DATA* subprograms superfluous. In addition, they can be used for packing procedures and their related data structures into libraries.

In VF90 these features may be used, in a restricted form, to define collections of (distributed) arrays, so-called *distribution modules*, which can then be shared by procedures. A module may contain type declaration statements, type definitions, distribution type definitions, processor array declarations, and interface blocks for external procedures. Modules have to be used to declare globally accessible processor arrays. Each program unit in which a global processor array is accessed must use the corresponding module.

To utilize data defined in a module, the *USE* statement must be specified. Using modules instead of explicit parameter passing makes programming easier, and in addition supports the compiler in generating efficient parallel code.

2.9 Local Procedures

In order to provide a mechanism for calling sequential procedures from a Vienna Fortran 90 program, a procedure may be specified with the keyword *LOCAL*. This mechanism is especially useful if some sequential standard algorithm should be applied to the local portion of a distributed object, independently on each processor. *LOCAL* procedures behave like ordinary sequential Fortran procedures[31] and it is assumed that they are executed on a single processor.

A local procedure must not contain any Vienna Fortran extensions, whatsoever. If a local procedure is executed on a particular processor, it may only access data that is owned by that processor. Besides that, it must not not produce any side effects. More precisely, it must not access or update a global variable or an object that is storage associated with a global variable or a subobject thereof, except through the use of a dummy argument.

In Vienna Fortran 90 a call to a local procedures must either appear inside of a forall loop or inside another local procedure. If a local procedure is called from a forall loop, the programmer is required to explicitly specify which processor should execute the procedure. This can be achieved by either appending an *ON*-clause to the call statement or by providing a work distribution specification in a surrounding forall loop.

It is the responsibility of the programmer to ensure that all the data accessed by a local procedure is exclusively local to the executing processor (i.e it must not be replicated). For this purpose Vienna Fortran 90 provides additional intrinsic functions. For example the Fortran 90 intrinsic function *LBOUND* may be used with a processor number as additional argument. In this case, it returns the smallest index of the corresponding array that has been mapped to that processor (see Appendix B).

[31] Local procedures also may be coded in any other sequential language like for example C.

Example 2.9.1 illustrates the use of local procedures in a parallel sorting application. In this example the subroutines *INTERCHANGE_SORT* and *MERGE* are ordinary sequential Fortran procedures which are used as *LOCAL* procedures.

Example 2.9.1: Local procedures

The program of Figure 2.3 sorts a one-dimensional array A which - when the process is completed - resides on a single processor. At the outset, the processor array R is declared. It is stipulated that it must contain at least $2^4 = 16$ *processors. If fewer processors are available at runtime, execution will not continue.*

The allocatable array A is declared with the keyword DYNAMIC, which indicates that its distribution may change during program execution. It is initially distributed by BLOCK, with the result that each processor in R receives M contiguous elements of A in its local memory. In the first FORALL loop the LOCAL procedure INTERCHANGE_SORT is called on each processor independently. The code in the sort subroutine INTERCHANGE_SORT does not require communication, since the array section passed in each call is local to the executing processor. Therefore, each processor independently sorts its local part of array A. Here, the programmer has made use of the knowledge that A has been distributed in a block-wise fashion.

Before the first call to the local subroutine MERGE, A is redistributed so that a contiguous block of 2M elements resides on every other processor. The two sorted halves of this block are merged, using a temporary array on each processor. The merged list of elements is written back to the array associated with A at the finish. Each time the second FORALL *loop is executed (L times in all), the lengths of the blocks of A are doubled, and the number of processors involved are halved. Hence, at program termination, the sorted array resides on the first processor. This merge routine too, is called with a particular data distribution in mind and no communication is required within it once the data has been moved to the appropriate processor.*

It is naturally not always desirable (or possible) to gather the elements of an array onto one processor in a sort and merge. Other approaches will require that the final list be distributed among all or a subset of the processors used can, however, be expressed just as easily using the features of VF90, since the user knows precisely which elements of the array are on any particular processor. ■

2.10 Discussion

We close the first part of the thesis with a discussion of the relationship between Vienna Fortran and Vienna Fortran 90. A major design goal of Vienna Fortran 90 was to extend the rich set of features provided by Vienna Fortran, which is based on Fortran 77, to Fortran 90. Whereas, the

```
PROGRAM SORT

   PROCESSORS, ASSERT (L>4) :: R(2**L)
   REAL, ALLOCATABLE, DISTRIBUTED(BLOCK), DYNAMIC :: A(:)

   READ(*,*) M; ALLOCATE(A(M*2**L)); READ(*,*) A

   FORALL (IP = 1, SIZE(R)) ON R(IP)
      CALL INTERCHANGE_SORT( A(LBOUND(A,1,IP) : UBOUND(A,1,IP)) )
   END FORALL

   K=1
   DO IM = 1, L
      K = K + K
      DISTRIBUTE(A(BLOCK) TO R(::K))
      FORALL (I = 1, SIZE(R), K) ON PROC(I)
         CALL MERGE( A(LBOUND(A,1,IP) : UBOUND(A,1,IP)) )
      END FORALL
   END DO

CONTAINS
   LOCAL SUBROUTINE INTERCHANGE_SORT(X)
      REAL X(:), TEMP
      INTEGER I,J

      DO I = 1, UBOUND(X) - 1
         DO J = I, UBOUND(X)
            IF (X(I) > X(J)) THEN
               TEMP = X(I); X(I) = X(J); X(J) = TEMP
            END IF
         END DO
      END DO
   END SUBROUTINE INTERCHANGE_SORT

   LOCAL SUBROUTINE MERGE(Y)
      REAL Y(:), TEMP(SIZE(Y))
      INTEGER HALF, I, I1, I2

      HALF = SIZE(Y)/2; I1 = 1; I2 = 1 + HALF
      DO I = 1, SIZE(Y)
         IF ( Y(I1) < Y(I2) ) THEN
            TEMP(I) = Y(I1); I1 = I1 + 1
            IF (I1 > HALF) THEN
               TEMP(I+1 : SIZE(Y)) = Y(I+1 : SIZE(Y)); EXIT
            END IF
         ELSE
            TEMP(I) = Y(I2); I2 = I2 + 1
            IF (I2 > SIZE(Y)) THEN
               TEMP(I+1 : SIZE(Y)) = Y(I+1 : HALF); EXIT
            END IF
         END IF
      END DO
      Y = TEMP
   END SUBROUTINE MERGE
END PROGRAM SORT
```

Figure 2.3: Parallel sort and merge

syntax (see Appendix A) has been redesigned in order to conform as much as possible with Fortran 90, most of the basic features were taken over from Vienna Fortran. Besides that, a number of features have a slightly modified semantics, have been deleted or clarified.

Whereas in Vienna Fortran all processor arrays of a program must have the same size, in Vienna Fortran 90 processor arrays of external procedures may have a size less than the processor arrays of the main program unit. Furthermore, reshapes of a processor array may be obtained by means of the Fortran 90 intrinsic function *RESHAPE*, and irregular processor subsets may be referenced by means of vector subscripts. The intrinsic distribution function *BLOCK(M)* has been added, and the concept of indirect distributions has been generalized such that it is possible to have multidimensional map arrays and to distribute individual array dimensions indirectly.

The implications of allocatable arrays in the context of alignment has been clarified. In Vienna Fortran 90 it is not allowed to align a statically allocated array with a dynamically allocated array, as it is not allowed to align a statically distributed array with a dynamically distributed array that does not have an initial distribution. The allocate statement has been modified such that it is possible to specify a distribution (or alignment) at the time of allocation.

Vienna Fortran 90 provides the *ALIGN* statement for aligning arrays at runtime whereas in Vienna Fortran both realignment and redistribution is achieved by means of the *DISTRIBUTE* statement. Furthermore, an array may only be realigned at runtime if no other objects are connected to it.

Whereas Vienna Fortran allows to specify only one-level connections, in Vienna Fortran 90 it is possible to specify multi-level connect relationships. It is however not allowed to connect an array to more than one primary array.

The Vienna Fortran mechanisms of user defined distribution functions and user defined alignment functions have been unified and generalized by introducing mapping procedures. Mapping procedures may be recursive, may be passed as arguments to mapping procedures and may also be used to specify arbitrary distributions of forall loop iterations to processor sections. Moreover, the concept of distribution type definitions has been extended and generalized.

Besides that, Vienna Fortran 90, since based on Fortran 90, provides many new features not found in Vienna Fortran, in particular distribution of derived types, distribution of pointer objects, modules for declaring globally accessible processor arrays and local procedures.

The relationship of Vienna Fortran 90 to other comparable languages, in particular HPF, is discussed in Chapter 7.

Part II

Compilation

Chapter 3

Model

In this section we define a formal model on which the compilation as well as the execution of Vienna Fortran 90 programs is based. The main part of the model provides concepts needed for the process of restructuring a VF90 program into a Fortran 90 message passing program. Many of these concepts are also applicable for the runtime support system which is needed for the execution of the parallel program on distributed memory multicomputers. In particular, we define formal concepts in order to model data distribution, ownership of data, work distribution, and communication.

The generated message passing program is based on the SPMD or data parallel programming model. In this model all processors execute a copy of the same program, which in the following will be referred to as node program. The node program is parameterized by the processor id, such that each processor executes the program on its own data. The process of transforming a VF90 program into a SPMD message passing program can be conceptually divided into three major steps.

1. distribution of the data space among the available processors
2. distribution of the computation among the processors
3. generation of necessary communication between the processors

In the following the information needed to implement these steps is formalized on the basis of *ordered sets*. In the next chapter we will focus on the actual computation of this information.

3.1 Basic Concepts

Definition 3.1.1: Data Space

> *The* **data space** *of a program is the set of data objects declared in the program and is denoted by $\mathcal{A}$. $\mathcal{A}$ contains two classes of objects:* **scalars** *and* **arrays**. *A scalar object has a* **rank** *of* 0. *The set of all elements in $\mathcal{A}$ is denoted by $\mathcal{E}$. Arrays designate structured sets of scalars and have a rank ≥ 1.* ■

Each declared array $A \in \mathcal{A}$ of rank n is associated with a standard index domain denoted by $\mathbf{I}^A = [L_1^A : U_1^A, \ldots, L_n^A : U_n^A]$ as defined in Section 2.2.1. The set of elements of A is denoted by $\mathcal{E}^A$.

The **set of processors**, $\mathbf{P}$, is represented in a program by one or more **processor arrays**. We will use the notational conventions introduced in Section 2.2.1; in particular, for a processor array R, $\mathbf{I}^R$ denotes the associated standard index domain, and $index^R : \mathbf{P} \rightarrow \mathbf{I}^R$ the function mapping processors to their index.

Any two processors in $\mathbf{P}$ communicate by exchanging messages. Our model abstracts from the machine topology, such as grid, torus, or hypercube and the related message passing mechanisms, and thus does not reflect different processor "distances". Furthermore, we do not take the number of real processors into account. Our model only considers "abstract processors" and relies on the run-time system to map abstract processors dynamically to the real processors.

3.2 Data Distribution

A distribution of an array A as defined in Section 2.2.1 is characterized by three components: the array index domain $\mathbf{I}^A$, the processor array index domain $\mathbf{I}^R$, and a distribution function δ_R^A which is a total function mapping array indices to non-empty sets of processor array indices. Note that this definition of a distribution allows describing any arbitrary mapping of an object to the available set of processors. An important class of distributions are the composite distributions where the distribution of a multidimensional array is described for each dimension separately.

Definition 3.2.1: Replication

An array $A \in \mathcal{A}$ is **(totally) replicated** *(with respect to R) iff* $\delta_R^A(\mathbf{i}) = \mathbf{I}^R,\ \forall \mathbf{i} \in \mathbf{I}^A$.
A is partially replicated iff $\exists\ \mathbf{i}\ s.t.\ |\delta_R^A(\mathbf{i})| > 1$. ■

The distribution and alignment specifications of a Vienna Fortran 90 program have to be processed in order to determine for each processor the set of elements that have to be stored in its local memory. This is achieved by evaluating on each processor the distribution function for each distributed object of the program.

Let in the following A denote an array that is distributed by means of a distribution function δ_R^A. The ordered set of indices $\mathbf{i} \in \mathbf{I}^A$ designating those elements that have to be allocated on a particular processor $p \in \mathbf{P}$ is referred to as local index set and defined as follows:

Definition 3.2.2: Local Index Set

Let A, $\mathbf{I}^A$, R, $\mathbf{I}^R$ and δ_R^A be given. The set of indices $\mathbf{i} \in \mathbf{I}^A$ of an array A which are mapped to a particular processor $p \in \mathbf{P}$ by means of a distribution function δ_R^A is called the **local index set** *of p with respect to A and denoted by $Local^A(p)$. It is defined by the total function*

$$Local^A : \mathbf{P} \rightarrow \mathcal{P}(\mathbf{I}^A)$$
$$\forall p \in \mathbf{P} : Local^A(p) = \{\mathbf{i} \in \mathbf{I}^A \mid index^R(p) \in \delta_R^A(\mathbf{i})\}$$

An array element $A(\mathbf{i})$ is said to be **owned** *by processor p iff $\mathbf{i} \in Local^A(p)$. The set of elements of A that are allocated in the local memory of a processor p is called the* **local segment** *of p with respect to A.* ∎

The local index set for a distributed array A has to be determined on each processor p and designates those elements of A that have to be allocated in the local memory of p. It is the basis for work distribution and communication generation.

3.3 Work Distribution

It is the task of the compiler to determine how to partition the work (i.e. computations) among the processors. The default mechanism for doing so is based on the owner computes rule. By applying the owner computes rule the work distribution is automatically derived from the specified data distribution. The owner computes paradigm implies that a processor only executes assignments to data it owns. In the following consider an assignment to an arbitrary array section of the form:

$$A(\vec{ref}) = ...$$

where $\vec{ref}$ denotes the left hand side **reference space** which corresponds to the extended index domain of the array section as defined in 2.2.1. The reference space with respect to a particular array section contains the indices of all elements that are referenced by that array section. The above array assignment statements is transformed in such a way that each processor $p \in \mathbf{P}$ only updates those elements $e \in \mathcal{E}^{A(\vec{ref})}$ that are owned by it. The subset of the reference space $\vec{ref}$ for which a particular processor p has to execute the assignment is called work index set of p and defined as follows:

Definition 3.3.1: Work Index Set

Consider an assignment to an array section $A(\vec{ref})$. The set of indices $\mathbf{i} \in \mathbf{I}^A$ for which element $A(\mathbf{i})$ has to be computed by processor $p \in \mathbf{P}$ is called the **work index set** *of processor p with respect to A and denoted by $Work^A(p)$. It is defined by a total function*

$$Work^A : \mathbf{P} \rightarrow \mathcal{P}(\mathbf{I}^A)$$

$$\forall p \in \mathbf{P} : \; Work^A(p) \;\; = \;\; \{\mathbf{i} \in \mathbf{I}^A \mid \mathbf{i} \in (Local^A(p) \cap \vec{ref})\}$$

∎

For the parallelization of assignment statements that reference distributed arrays each processor has to determine the set of non-local elements it must receive from other processors in order to perform the computations defined by its work index set. Furthermore, it has to determine the set of local elements that have to be sent to other processors. This leads us to the definition of communication sets.

3.4 Communication

Communication sets determine those elements of a particular right hand side reference of an assignment statement a processor p must send (or receive) to (from) another processor q, in order to perform the computations defined by its work index set. Consider an array assignment of the form

$$A(\vec{lhs}) = B(\vec{rhs})$$

where $\vec{lhs}$ and $\vec{rhs}$ denote the left hand side and right hand side reference space, respectively. Let f denote the total function that maps an index of the left hand side reference to its corresponding index of the right hand side reference.

Definition 3.4.1: Send Index Set

The set of indices $\mathbf{i} \in \mathbf{I}^B$ *for which elements* $B(\mathbf{i})$ *have to be sent from processor* p *to processor* q *is called the* **send index set** *of processor* p *to* q *with respect to* B *and denoted by* $Send^B(p,q)$ *. It is defined by a total function*

$$Send^B : \mathbf{P} \text{ x } \mathbf{P} \to \mathcal{P}(\mathbf{I}^B)$$

$$\forall p,q \in \mathbf{P} : \; Send^B(p,q) \;\; = \;\; \{\mathbf{i} \in \mathbf{I}^B \mid \mathbf{i} \in (f(Work^A(q)) \cap Local^B(p))\}$$

■

Definition 3.4.2: Receive Index Set

The set of indices $\mathbf{i} \in \mathbf{I}^B$ *for which elements* $B(\mathbf{i})$ *have to be received by processor* p *from processor* q *is called the* **receive index set** *of processor* p *from* q *with respect to* B *and denoted by* $Recv^B(p,q)$ *. It is defined by a total function*

$$Recv^B : \mathbf{P} \text{ x } \mathbf{P} \to \mathcal{P}(\mathbf{I}^B)$$

$$\forall p,q \in \mathbf{P} : \; Recv^B(p,q) \;\; = \;\; \{\mathbf{i} \in \mathbf{I}^B \mid \mathbf{i} \in (f(Work^A(p)) \cap Local^B(q))\}$$

■

In order to implement accesses to non-local data, it is necessary to allocate on each processor storage for all elements that have to be received from other processors. One way of managing non-local data is to allocate temporary buffers. As a consequence, the work index set has to be split into those references that are purely local and into references that access non-local data which has been received in a temporary buffer. Therefore, we have to split a processors work index set into two disjoint subsets.

The subset of $Work(p)$ that corresponds to all references that can be processed without communication is called local-work index set. The other subset, called non-local work index set, contains all references that require access to non-local data.

Definition 3.4.3: Local-Work Index Set

The set of indices $\mathbf{i} \in \mathbf{I}^A$ *for which the assignment to* $A(\mathbf{i})$ *can be executed without communication is called the local-work index set of processor* p *with respect to* A *and denoted by* $Local\_Work^A(p)$. *It is defined by a total function:*

$$Local\_Work^A : \mathbf{P} \to \mathcal{P}(\mathbf{I}^A)$$

$$\forall p \in \mathbf{P}: \; Local\_Work^A(p) \;=\; \{\mathbf{i} \in \mathbf{I}^A \mid \mathbf{i} \in f^{-1}(Send^B(p,p))\}$$

∎

Definition 3.4.4: Non-Local-Work Index Set

The set of indices $\mathbf{i} \in \mathbf{I}^A$ *for which the assignment to* $A(\mathbf{i})$ *requires communication is called the non-local work index set of processor* p *with respect to* A *and denoted by* $Nonlocal\_Work^A(p)$. *It is defined by a total function:*

$$Nonlocal\_Work^A : \mathbf{P} \to \mathcal{P}(\mathbf{I}^A)$$

$$\forall p \in \mathbf{P}: \; Nonlocal\_Work^A(p) \;=\; \{\mathbf{i} \in \mathbf{I}^A \mid \mathbf{i} \in (Work^A(p) \backslash f^{-1}(Local^B(p))\}$$

∎

Using these last two sets, an array assignment statement can be transformed in such a way that after computation of the send set each processor executes a send operation, followed by the execution of its local work set. When finished with the local computations non-local data is received and the operations defined by the non-local work set are performed. The implementation of an array assignment statement $A(\vec{hs}) = B(r\vec{hs})$ on a processor p therefore can be sketched as follows:

1. generate $Send^B(p,q)$ for all $q \in \mathbf{P}, q \neq p$
2. perform send operations
3. perform computations defined by $Local\_Work^A(p)$
4. generate $Recv^B(p,q)$, for all $q \in \mathbf{P}, q \neq p$
5. perform receive operation
6. perform computations defined by $Nonlocal\_Work^A(p)$

As can be seen, this execution scheme results in overlapping of communication and computation and thus permits to hide the latency of remote accesses. The fetch-before-store semantics of array assignment statements warrants the correctness of this scheme.

Another important technique for handling accesses to non-local data is the overlap concept [51]. With this method non-local array elements are not stored in separate buffers but in the overlap area surrounding the local segment of an array.

The **overlap area** of an array A with respect to a processor p, is the smallest rectilinear contiguous area around the local segment of A with respect to the processor p, containing all non-local variables accessed. The overlap area for an array A with respect to a processor p is specified by its **overlap description**, $od^A(p)$, which is determined by the maximum offsets for every dimension of the local segment of p with respect to A.

For an n-dimensional array A, this takes the form

$$od^A(p) = [o_{l_1} : o_{r_1}, \ldots, o_{l_n} : o_{r_n}]$$

where, o_{l_i} and o_{r_i} denote the offsets with respect to the lower and upper bound of dimension i.

Handling non-local accesses by means of overlaps may significantly improve the organization of communication; it facilitates memory allocation and the local addressing of arrays. A major advantage of the overlap concept is that it is not necessary to split local and non-local references. The drawback of this scheme is that it is not generally applicable. The applicability of the overlap concept in the context of the basic intrinsic distributions and its limitations are discussed in greater detail in section 4.2.4.6.

3.5 Index Conversion

Each processor has to allocate memory for those elements of distributed objects that are owned by it. In order to correctly access the local data allocated for a distributed array on a particular processor, it is necessary to convert global indices of the original array into offsets of the local segment. In order to optimize storage, an index conversion function has to compute for every index the number of smaller indices that have been mapped to the same processor. Index conversion is discussed in more detail in Section 4.2.2.

3.6 Composite Distributions

A composite distribution as defined in Section 2.4.1.1 describes the distribution of each array dimension separately. In case of a composite distribution of an array A the local index sets of two different processors are either disjoint or identical. More precisely: Assume that the set of processors is given by $\mathbf{P} = \{p_1, \ldots, p_r\}$, then

$$\forall i, j,\ 1 \leq i, j \leq r,\ \ Local^A(p_i) \cap Local^A(p_j) \neq \phi \ \Rightarrow\ Local^A(p_i) = Local^A(p_j)$$

Note that this condition might not be satisfied in case of a general distribution.

The following sections are restricted to composite distributions where arrays may be distributed directly or implicitly and each array dimension may have a block, general-block, cyclic or block-cyclic distribution or it is replicated. These basic intrinsic distribution functions map each index of an array dimension to exactly one index of a processor array dimension. Therefore, they can be characterized by a total function $\delta_{R_j}^{A_i} : \mathbf{I}_i^A \rightarrow \mathbf{I}_j^R$ from index dimension i, $1 \leq i \leq rank(A)$, of array A to index dimension j, $1 \leq j \leq rank(R)$, of the processor array R.

Moreover, alignment functions will be restricted to total functions $\alpha_{A_i}^{B_k} : \mathbf{I}_k^B \rightarrow \mathbf{I}_i^A$ from a dimension k, $1 \leq k \leq rank(B)$, of an array B to dimension i, $1 \leq i \leq rank(A)$, of another array A. Under these restrictions the distribution function of the $k-th$ dimension of B with respect to the $i-th$ dimension of the processor array R is given by $\delta_{R_j}^{B_k} = \delta_{R_j}^{A_i} \circ \alpha_{A_i}^{B_k}$.

Chapter 4

Source to Target Mapping

4.1 Introduction

In the following sections we show how Vienna Fortran 90 programs are automatically transformed into data parallel programs based on the SPMD programming model. We first describe a subset of Vienna Fortran 90 which is treated in the remainder of this thesis. After that, we focus on the basic elements needed for the parallelization of Vienna Fortran 90 programs. In particular, we show how the computation of local index sets, work sets and communication sets, as defined in Chapter 3, is realized. Finally, we specify the mapping of selected Vienna Fortran 90 constructs to Fortran 90 message passing code.

4.1.1 Source Language

The compilation as described in the remainder of this work is restricted to the following subset of Vienna Fortran 90.

- Vienna Fortran Constructs:
  Data distribution and alignment specifications are restricted to static distributions only. However, many of the presented concepts are also applicable in the context of dynamic data distribution. Each array dimension may be either distributed by means of the intrinsic distributions functions $BLOCK[(M)]$ or $CYCLIC[(M)]$ or it may be replicated. Distributions may be specified with respect to regular processor sections. Programs may be parameterized by the number of processors. Alignment functions may be arbitrary linear functions.

- Fortran 90 Constructs:
  The main part of this work is based on the parallelization of the data parallel Fortran 90 constructs, in particular array assignments and the WHERE statement. However, many of the techniques are also applicable for DO loops. Furthermore, arrays may be allocatable or automatic. Dummy arrays may be either assumed shape arrays or explicit shape arrays. They may either be distributed explicitly or inherit the distribution from the corresponding actual argument.

4.1.2 Target Language

The output of the parallelization process is a Fortran 90 SPMD augmented with calls to a message passing library.

4.2 Basic Elements

In this section we focus on the computation of data distribution, work distribution and communication. If not stated otherwise, we assume that Vienna Fortran 90 programs are parallelized for a parameterized number of processors. Moreover, distributions may be runtime dependent. A distribution of an array A is runtime dependent if any of the following conditions is satisfied:

- A is dynamically allocated
- the corresponding distribution function depends on run-time data
- the size of the processor section to which A is distributed is unknown at compile time
- A cannot be bound to a particular distribution at compile time, as it may be the case for dummy arrays with assumed distribution

As a consequence, the execution of Vienna Fortran 90 programs requires significant run time support.

In the following we give an overview of the functionality that has to be provided be the Vienna Fortran runtime support system.

- Data Organization Functions
  These functions determine dynamically the local index set of a distributed object on a particular processor and are therefore a prerequisite for the allocation of distributed objects. Furthermore, the runtime system must contain functions that realize the conversion of the original global indices of a distributed array into a processors local indices.
- Work Distribution Functions
  Based on the actual distribution of a distributed object each processor usually only performs assignments to those parts of the object that are allocated in its local memory. For example, in the case of an array assignment statement each processor has to determine for which elements of the left hand side array it has to execute the assignment.
- Support for Communication Generation
  Usually it will not be possible to determine at compile time the communication that is necessary in order to perform for example an array assignment statement. Therefore, each processor has to determine at runtime which data elements it has to send to or receive from other processors in order to guarantee correct program execution. The runtime system also contains functions to handle buffering of non-local data.

In order to realize the above mentioned functions it is necessary to generate a runtime representation of distributed objects, which will be referred to as *layout descriptors*. Layout descriptors (see

Section 4.2.6) contain all information that is needed to determine an array's distribution at runtime.

The remainder of Section 4 is organized as follows. In the next subsection we specify the computation of local index sets and discuss how the local index set of a distributed array can be represented. Section 4.2.2 addresses memory management for distributed objects and techniques for index conversion. Computation of work sets and communication sets is treated in Section 4.2.3 and Section 4.2.4, respectively.

4.2.1 Computation of Local Index Sets

In this section we present formulas for the computation of local index sets of distributed objects.[1] The local index set of an array can be determined at compile time only if all arguments of the corresponding distribution function, the number of processors on which the program will execute and the array size is known at compile time. In all other cases the computation of the local index set has to be deferred until run time and as a consequence all statically allocated objects have to be transformed into dynamically allocated objects. Usually the local index set of a distributed object is determined every time an allocation instance is initiated by means of functions contained in the runtime library.

Local index sets are a prerequisite for the allocation of distributed objects, the computation of work distribution and communication generation. Moreover, the local index set of an array A is needed for the computation of local index sets for arrays that are aligned with A.

We now derive from the Vienna Fortran distribution and alignment specifications closed form expressions (analog to Fortran 90 section notation) for the computation of local index sets. Depending on whether an array is directly distributed or implicitly distributed (by means of alignment) different formulas are derived. These formulas can be used either in the compiler, or in cases mentioned above, they are incorporated into functions contained in the runtime library.

Parameter Definitions and Assumptions:

The parameters in the formulas presented in the remainder of this section have the following meaning.

A	...	directly distributed one-dimensional array
N_A	...	size of array A
B	...	one-dimensional array that is aligned with A
N_B	...	size of array B
M	...	parameter for *BLOCK(M)* or *CYCLIC(M)* distributions
P	...	total number of processors in dimension
p	...	index of individual processor, $1 \leq p \leq P$
$\alpha(i)$	...	alignment function $\alpha(i) = i.k + d$
$[l : u : s]$	...	array section

[1]In the following we assume that data objects for which no distribution has been specified are replicated over all processors. In this case, the local index set on each processor corresponds to the whole array.

If the array is implied by the context, the subscript for N will be omitted. In the following we assume that all declarations of objects (i.e. arrays, processor arrays) have been normalized such that the lower bound in each dimension is 1. Computation of all sets is specified using *global indices* rather than local ones. In Section 4.2.2 we define functions that convert between global and local indices. The formulas presented in the next sections are restricted to one dimensional arrays. The local index set of a multidimensional array is constructed via the Cartesian product of each dimension as described in Section 4.2.5.

4.2.1.1 Local Index Set of a Directly Distributed Array

Due to the nature of the Vienna Fortran intrinsic distributions (*BLOCK*, *CYCLIC*, *BLOCK(M)* and *CYCLIC(M)*), the local index set of arrays can be represented as either a single regular section or as a union of regular sections by means of triplet notation. In the following we call the subset of an array's local index set described by a single regular section an **index section**.

The local index set of an array with one of the above intrinsic distributions has the following properties:

1. For *BLOCK*, *CYCLIC* and *BLOCK(M)* distributed dimensions the local index set can be described by a single index section.
2. For all distributions except for *CYCLIC*, the local index set can be described by contiguous index sections. (i.e. with stride 1).

Note that property (1) is usually not fulfilled in the case of *CYCLIC(M)* distributions, where the local index set of a processor may consist of more than one index section. This complicates the handling of these sets at compile- and/or at runtime. In the following we derive from the distribution function δ the local index set of an array in terms of index sections.

4.2.1.1.1 *BLOCK* Distribution

The distribution function δ for an array $A(1:N)$ in case of a $BLOCK$ distribution with respect to P processors is defined as follows: Let $q := \left\lfloor \frac{N}{P} \right\rfloor$, $r := (N \ mod \ P)$, and $1 \leq i \leq N$:

$$\delta(i) = \begin{cases} \left\{ \left\lceil \frac{i}{q+1} \right\rceil \right\} & \text{iff } 1 \leq i \leq (q+1).r \\ \\ \left\{ \left\lceil \frac{i-r}{q} \right\rceil \right\} & otherwise \end{cases} \tag{4.1}$$

As a consequence, the size of blocks allocated on different processors may differ at most by one. The first r processors get a block of size $\left\lceil \frac{N}{P} \right\rceil$ and the remaining $P - r$ processors get a block of length $\left\lfloor \frac{N}{P} \right\rfloor$. The local index set on a particular processor p specified in triplet notation is therefore given by:

$$Local^A(p) = \begin{cases} \left[(p-1).\left\lceil \frac{N}{P} \right\rceil + 1 : p.\left\lceil \frac{N}{P} \right\rceil : 1\right] & \text{iff } 1 \leq p \leq N \ mod \ P \\ \\ \left[r + (p-1).\left\lfloor \frac{N}{P} \right\rfloor + 1 : r + p.\left\lfloor \frac{N}{P} \right\rfloor : 1\right] & \text{otherwise} \end{cases} \tag{4.2}$$

In the case where N is exactly divisible by P the local index set is identical to those generated by a $BLOCK(N/P)$ distribution as described next.

4.2.1.1.2 $BLOCK(M)$ Distribution

For *BLOCK(M)* distributions, where the block size is specified explicitly, the distribution function δ is defined as follows:

$$\delta(i) = \left\{ \left\lceil \frac{i}{M} \right\rceil \mid M \geq \left\lceil \frac{N}{P} \right\rceil \right\},\ 1 \leq i \leq N \tag{4.3}$$

The set of elements of array $A(1:N)$ owned by processor p, $Local^A(p)$, therefore is given by the index section:

$$Local^A(p) \quad = \quad [(p-1).M + 1 : min(p.M, N) : 1] \tag{4.4}$$

Note that $M \geq \left\lceil \frac{N}{P} \right\rceil$ must be satisfied.

4.2.1.1.3 $CYCLIC$ Distribution

The distribution function in case of a *CYCLIC* distribution is given by

$$\delta(i) = \{1 + (i-1)\ mod\ P\},\ 1 \leq i \leq N \tag{4.5}$$

The local index set on a particular processor p always can be represented with a single, however non-contiguous, triplet, which is given by:

$$Local^A(p) = [p : N : P] \tag{4.6}$$

4.2.1.1.4 $CYCLIC(M)$ Distribution

Block-cyclic or CYCLIC(M) distributions are a generalization of the block and cyclic distributions. As a consequence, a block-cyclic distribution may be interpreted either as a series of block distributions or as a series of cyclic distributions. The distribution function in case of a *CYCLIC(M)* distribution is given by :

$$\delta(i) \quad = \quad \left\{1 + \left\lfloor \frac{i-1}{M} \right\rfloor\ mod\ P\right\} = \left\{1 + \left\lfloor \frac{(i-1)\ mod\ (P.M)}{M} \right\rfloor\right\},\ 1 \leq i \leq N \tag{4.7}$$

The set of elements owned by a processor p might no longer be a single index section. If $M < \left\lceil \frac{N}{P} \right\rceil$, the local index set of some processors will consist of a set of index sections. The local index set on a particular processor p in case of a $CYCLIC(M)$ distribution can be described either as a union of index sections generated by a $BLOCK(M)$ distribution or as a union of index sections generated by a $CYCLIC(1)$ distribution. This is illustrated in Figure 4.1.

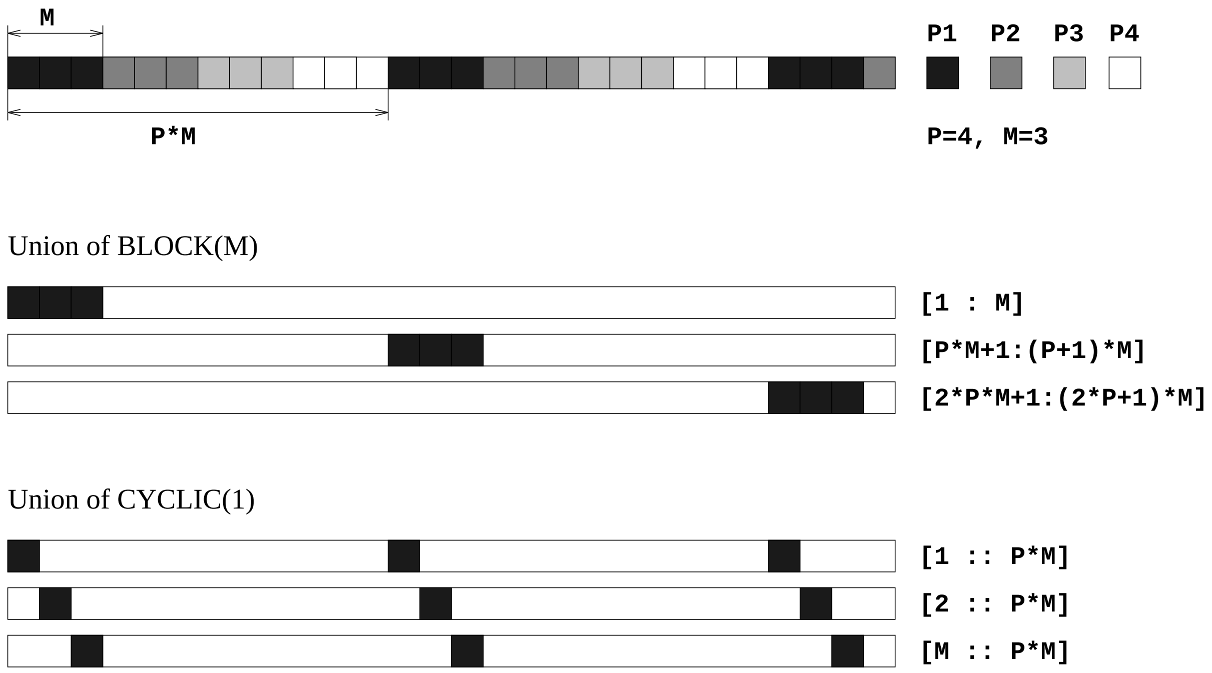


Figure 4.1: Representation of *CYCLIC(M)* distributions

The set of local indices of an array $A(1:N)$ with a block-cyclic distribution *CYCLIC(M)* on a particular processor p can be represented as a union of n_p contiguous index sections, denoted by σ_j^A, as follows:

$$Local^A(p) \;=\; \bigcup_{j=1}^{n_p} \sigma_j^A(p) = \bigcup_{j=1}^{n_p} \left[\mathcal{L}_j^A(p) : \mathcal{U}_j^A(p) : 1\right] \tag{4.8}$$

where,

$$n_p \;=\; \left\lceil \frac{N-(p-1).M}{P.M} \right\rceil \tag{4.9}$$

$$\mathcal{L}_j^A(p) \;=\; (p-1).M + 1 + (j-1).P.M \tag{4.10}$$

$$\mathcal{U}_j^A(p) \;=\; min(\mathcal{L}_j^A(p) + M - 1, N) \tag{4.11}$$

In the following we will refer to this representation as **block representation**.

If, alternatively, the local index set is represented as a union of M non-contiguous index sections, denoted by $\hat{\sigma}_j^A$, with stride $P.M$, we have:

$$Local^A(p) \;=\; \bigcup_{j=1}^{M} \hat{\sigma}_j^A = \bigcup_{j=1}^{M} [(p-1).M + j : N : P.M] \tag{4.12}$$

This representation is called **cyclic representation**.

Which of the both interpretations is more suitable mainly depends on the actual values of M and P. If for example $N > P.M^2$, then $n_p > M$ and the block representation consists of more index sections than the cyclic representation.

In order to derive a compact representation for the set of local elements, which can be used by the compiler and/or run-time system, we can utilize the fact that $\mathcal{L}_{i+1}^A(p) - \mathcal{L}_i^A(p) = P.M$. As a consequence, the local index set of a particular processor can be represented by means of four parameters. For that purpose we introduce the *extended section notation* which can be interpreted as a union of $BLOCK(M)$ distributions or a union of $CYCLIC(1)$ distributions as follows:

1. Representation of $CYCLIC(M)$ as union of $BLOCK(M)$

$$\langle L : U : S : M \rangle_B = \bigcup_{i=1}^{\lceil \frac{U-L}{S} \rceil} [L + (i-1).S : min(U, L + (i-1).S + M - 1) : 1] \qquad (4.13)$$

2. Representation of $CYCLIC(M)$ as union of $CYCLIC(1)$

$$\langle L : U : S : M \rangle_C = \bigcup_{i=1}^{M} [L + i - 1 : U - M + i : S] \qquad (4.14)$$

Using this extended section notation the local index set of an array with a block-cyclic distribution can be represented in the following manner:

$$Local^A(p) \quad = \quad \langle \mathcal{L}(p) : \mathcal{U}(p) : P.M : M \rangle \qquad (4.15)$$

where

$$\mathcal{L}(p) \quad = \quad (p-1).M + 1 \qquad (4.16)$$
$$\mathcal{U}(p) \quad = \quad min(N, (p + (n_p - 1).P).M) \qquad (4.17)$$

and n_p is defined as in (4.9).

4.2.1.2 Local Index Set of an Aligned Array

The local index set of an array B that is aligned with an array A can only be determined if the local index set of A is already known. In the following we assume that the alignment function[2] is linear and of the form $\alpha(i) = k.i + d$, $1 \leq i \leq N_B$.

Alignment can be specified in various forms (see Section 2.4.2.1). For example the following alignment directives have the same meaning:

[2] We assume that $k \neq 0$.

```
PARAMETER (K=..., D=...)
REAL,DIMENSION(1:N) :: A(1:K*N+D),B

ALIGNED(B(I) WITH A(K*I+D)) :: B
ALIGNED(A(K+D:K*N+D:K))     :: B
```

We base our formulas for aligned arrays on the first form of the above example where the alignment function is explicit in the declaration. In case of a section-based alignment specification the alignment function can be determined as described on page 40.

As described in Section 2.4.2.1 the alignment relationship between a set of arrays can be represented by means of an alignment tree. A path from a node in the alignment tree to the root is called **alignment chain**. The root of an alignment tree is called alignment basis and has to be distributed directly. An immediate alignment tree (i.e. a tree of height > 1) is transformed by the compiler into an ultimate alignment tree (i.e. a tree of height 1).

ALIGNED(B_1(I) WITH A(e_1(I))) :: B_1
ALIGNED(B_2(I) WITH B_1(e_2(I))) :: B_2
...
ALIGNED(B_n(I) WITH B_{n-1}(e_n(I))) :: B_n

ALIGNED(C_1(I) WITH A(c_1(I))) :: C_1
ALIGNED(C_2(I) WITH C_1(c_2(I))) :: C_2
ALIGNED(C_3(I) WITH C_1(c_3(I))) :: C_3

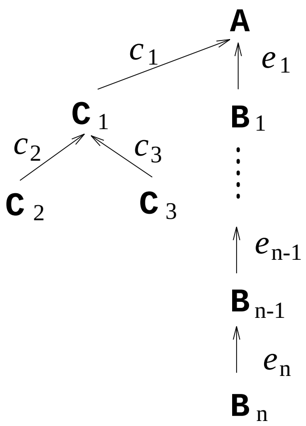


Figure 4.2: Alignment Trees

Consider the alignment specifications for arrays B_i, $1 \leq i \leq n$ of Figure 4.2. The local index set of an array B_i, $1 \leq i \leq n$, with respect to the alignment basis A can be described by means of the composed alignment function $e'_i = e_1 \circ e_2 \ldots \circ e_i$ and the local index set of A. Similarly the alignment relationship between arrays B_i and B_j, $i > j$, is determined by means of the composed alignment function $e'_i = e_j \circ e_{j+1} \circ \ldots \circ e_i$.

The set of local indices on processor p of an array $B(1 : N_B)$, which is ultimately aligned with an array A is denoted by $Local^B_A(p)$. The distribution function of array B can be expressed as follows:

$$\delta^B_A = \delta^A \circ \alpha^B_A \tag{4.18}$$

The local index set of B can be determined by intersection of the local index set of the alignment source array A on which the inverse alignment function α^{-1} has been applied, with the index domain of array B.

$$Local_A^B(p) \quad = \quad \alpha^{-1}(Local^A(p) \cap \alpha([1:N_B])) \quad = \quad \alpha^{-1}(Local^A(p)) \cap [1:N_B] \qquad (4.19)$$

In the next section we specify formulas for the computation of the intersection of two array sections which are a prerequisite for the computation of local index sets of aligned arrays. They are also the basis for the computation of work sets and communication sets, as described later. Except for sections with negative stride these formulas are similar to those described in [79].

4.2.1.2.1 Regular Section Intersection

The intersection of two regular array sections is defined as follows:

If both sections are unstrided

$$[l_1 : u_1] \cap [l_2 : u_2] \quad = \quad [max(l_1, l_2), min(u_1, u_2)] \qquad (4.20)$$

If one section contains a stride $s > 0$

$$[l_1 : u_1] \cap [l_2 : u_2 : s] \quad = \quad [max(next(l_1, l_2, s), l_2) : min(u_1, u_2) : s] \qquad (4.21)$$

If both sections have strides, $s_1 > 0, s_2 > 0$

$if \ (l_1 \equiv l_2(\text{mod gcd}(s_1, s_2)))$

$$[l_1 : u_1 : s_1] \cap [l_2 : u_2 : s_2] \quad = \quad [first(l_1, s_1, l_2, s_2) : min(u_1, u_2) : lcm(s_1, s_2)] \qquad (4.22)$$

$otherwise$

$$[l_1 : u_1 : s_1] \cap [l_2 : u_2 : s_2] \quad = \quad \phi \qquad (4.23)$$

The function $n = next(l_1, l_2, s)$ finds the smallest element n of a section $[l_2 :: s]$ such that $n \geq l_1$ and $n = l_2 \ mod \ s$. It can be computed as follows:

$$next(a, b, c) \quad = \quad a + (b - a) \ mod \ c \qquad (4.24)$$

In the case where both sections have a positive stride function $first$ determines the first element of the intersection. In order to compute $first(l_1, s_1, l_2, s_2)$, where $s_1 > 0, s_2 > 0$, we determine by means of Euclid's extended gcd algorithm [138] integers n_1, n_2 such that $s_1.n_1 + s_2.n_2 = gcd(s_1, s_2)$.[3] The function $first$ then can be computed as follows:

$$first(l_1, s_1, l_2, s_2) \quad = \quad next\left(max(l_1, l_2), \frac{s_1 n_1 (l_2 - l_1)}{gcd(s_1, s_2)} + l_1, lcm(s_1, s_2)\right) \qquad (4.25)$$

Note that so far we have considered only the cases where both sections have a positive stride. If one of the involved sections has a negative stride, the above formulas cannot be applied since the extended gcd algorithm is only defined for positive stride values. In case of a negative stride we must

[3] A solution exists iff $l_1 = l_2 mod(gcd(s_1, s_2))$.

slightly modify the above formulas because the role of the upper bound and lower bound is reversed.

For this purpose we define two additional functions *rev* and *last*. Function *rev* reverses a regular section by interchanging the upper bound and the lower bound, and switching the sign of the stride. However, this could lead to a section which does not have the same elements as the original section, since the upper bound must not necessarily be a section member as for example in $[1:10:2]$. Therefore, function *last*, which determines the last member of a section, is applied prior to *rev* in order to determine the *true* upper bound of the section.

If $s_1 < 0$ and $s_2 > 0$

$$[l_1 : u_1 : s_1] \cap [l_2 : u_2 : s_2] = rev(rev([l_1 : u_1 : s_1]) \cap [l_2 : u_2 : s_2]) \tag{4.26}$$

If $s_1 < 0$ and $s_2 < 0$

$$[l_1 : u_1 : s_1] \cap [l_2 : u_2 : s_2] = rev(rev([l_1 : u_1, s_1]) \cap rev([l_2 : u_2 : s_2])) \tag{4.27}$$

where

$$rev([l : u : s]) = [last(l, u, s) : l : -s] \tag{4.28}$$
$$last(l, u, s) = u - sign(s).(abs(u - l) \ mod \ abs(s)) \tag{4.29}$$

4.2.1.2.2 Alignment Source with $BLOCK(M)$ Distribution

If an array $B(1 : N_B)$ is aligned with an array $A(1 : N_A)$, that has a *BLOCK[(M)]* distribution, by means of the alignment function $\alpha(i) = i.k + d$, then the local index set of A on a particular processor p is given by:

Case 1: $k > 0$

$$\begin{aligned} Local_A^B(p) &= \left[max\left(1, \left\lceil \alpha^{-1}\left(\mathcal{L}^A(p)\right)\right\rceil\right) : min\left(N_B, \left\lfloor \alpha^{-1}\left(\mathcal{U}^A(p)\right)\right\rfloor\right) : 1\right] \\ &= \left[max\left(1, \left\lceil \frac{\mathcal{L}^A(p) - d}{k}\right\rceil\right) : min\left(N_B, \left\lfloor \frac{\mathcal{U}^A(p) - d}{k}\right\rfloor\right) : 1\right] \end{aligned} \tag{4.30}$$

Case 2: $k < 0$

$$\begin{aligned} Local_A^B(p) &= \left[max\left(1, \left\lceil \alpha^{-1}\left(\mathcal{U}^A(p)\right)\right\rceil\right) : min\left(N_B, \left\lfloor \alpha^{-1}\left(\mathcal{L}^A(p)\right)\right\rfloor\right) : 1\right] \\ &= \left[max\left(1, \left\lceil \frac{\mathcal{U}^A(p) - d}{k}\right\rceil\right) : min\left(N_B, \left\lfloor \frac{\mathcal{L}^A(p) - d}{k}\right\rfloor\right) : 1\right] \end{aligned} \tag{4.31}$$

where $\mathcal{L}^A(p)$ and $\mathcal{U}^A(p)$ denote the bounds of $Local^A(p)$ as defined in Formula 4.2 and 4.4, respectively.

4.2.1.2.3 Alignment Source with $CYCLIC$ Distribution

In case of a $CYCLIC$ or $CYCLIC(1)$ distribution of the alignment source array the local index set $Local_A^B(p)$ still can be represented by means of a single index section as follows:

$$Local_A^B(p) \quad = \quad \begin{cases} \left[\mathcal{L}(p) : \mathcal{U}(p) : \left\lfloor \frac{lcm(P,k)}{k} \right\rfloor\right] & if\ d = p\ mod\ gcd(k,P) \\ \phi & otherwise \end{cases} \tag{4.32}$$

Case 1: $k > 0$

$$\mathcal{L}(p) \quad = \quad \left\lfloor \frac{first\,(p, P, d+k, k) - d}{k} \right\rfloor \tag{4.33}$$

$$\mathcal{U}(p) \quad = \quad min\left(\left\lfloor \frac{N_A - d}{k} \right\rfloor, N_B \right) \tag{4.34}$$

Case 2: $k < 0$

$$\mathcal{L}(p) \quad = \quad last\left(\mathcal{U}(p), \left\lfloor \frac{min(N_B, l) - d}{k} \right\rfloor, -\mathcal{S}(p) \right) \tag{4.35}$$

$$\mathcal{U}(p) \quad = \quad \left\lfloor \frac{first(p, P, N.k + d, abs(k)) - d}{k} \right\rfloor \tag{4.36}$$

Note that a particular processor p has a non-empty local index set only in the case where the condition $d = p\ mod\ gcd(k, P)$ is satisfied.

4.2.1.2.4 Alignment Source with $CYCLIC(M)$ Distribution

Let $Local^A(p)$ denote the local index set of a *CYCLIC(M)*-distributed alignment basis A on a particular processor p. According to formula 4.8 it can be represented by a union of contiguous index sections. The local index set of an array $B(1 : N_B)$ which is aligned with A by the alignment function $\alpha(i) = i.k + d$ can be computed according to Formula 4.19 as follows:

Case 1: $k > 0$

$$Local_A^B(p) = \bigcup_{j=1}^{n_p} \sigma_j^B(p) = \bigcup_{j=1}^{n_p} \left[\mathcal{L}_j^B(p) : \mathcal{U}_j^B(p) : 1\right] \tag{4.37}$$

$$\mathcal{L}_j^B(p) \quad = \quad max\left(1, \left\lceil \alpha^{-1}\left(\mathcal{L}_j^A(p)\right)\right\rceil\right) \quad = max\left(1, \left\lceil \frac{\mathcal{L}_j^A(p) - d}{k} \right\rceil\right) \tag{4.38}$$

$$\mathcal{U}_j^B(p) \quad = \quad min\left(N_B, \left\lfloor \alpha^{-1}\left(\mathcal{U}_j^A(p)\right)\right\rfloor\right) = min\left(N_B, \left\lfloor \frac{\mathcal{U}_j^A(p) - d}{k} \right\rfloor\right) \tag{4.39}$$

Case 2: $k < 0$

$$Local_A^B(p) = \bigcup_{j=1}^{n_p} \sigma_j^B(p) = \bigcup_{j=1}^{n_p} \left[\mathcal{L}_j^B(p) : \mathcal{U}_j^B(p) : -1\right] \tag{4.40}$$

$$\mathcal{L}_j^B(p) = min\left(N_B, \left\lfloor \alpha^{-1}\left(\mathcal{L}_j^A(p)\right)\right\rfloor\right) = min\left(N_B, \left\lfloor \frac{\mathcal{L}_j^A(p) - d}{k} \right\rfloor\right) \tag{4.41}$$

$$\mathcal{U}_j^B(p) = max\left(1, \left\lceil \alpha^{-1}\left(\mathcal{U}_j^A(p)\right)\right\rceil\right) = max\left(1, \left\lceil \frac{\mathcal{U}_j^A(p) - d}{k} \right\rceil\right) \tag{4.42}$$

This representation of the local index set is called block representation. Note that the number of index sections of $Local_A^B(p)$ in the above formulas is equal to the number of index sections of the local index set $Local^A(p)$ of the alignment source array A. However, depending on the parameters of the alignment function some index sections at the beginning or at the end may be empty. Moreover, the local index set $Local_A^B(p)$ usually cannot be represented by means of the extended section notation as described on page 96. The reason for this is that the number of elements in different index sections usually differs. The implications of $CYCLIC(M)$ distributions in conjunction with alignment and an alternative cyclic representation of the local index set is discussed in more detail in Section 4.2.2.6.

4.2.2 Index Conversion and Memory Management

In the previous sections we have described how the local index set of a distributed array can be determined on a particular processor. The formulas presented were defined with respect to the original global indices and allow each processor of the parallel program to determine which indices of a distributed array it owns.

In order to generate an SPMD program, each processor has to determine how to allocate distributed arrays. The memory allocation scheme determines if and how global references of the source program have to be translated into local references.

Another aspect of memory allocation is the close interrelation with the question of how to manage storage for non-local data accesses. For example in the case of overlaps the local segment is extended to keep values exchanged during overlap communication. In the case where non-local data elements are managed via buffers or hash tables, it is necessary to split local and non-local references of a distributed object. Management of non-local data will be discussed later.

In the following we describe a memory allocation model which allows to allocate minimal space for a distributed array on a particular processor. Each processor only allocates those elements of distributed arrays that have been mapped to it by means of the user specified distribution directives. As a consequence, global references of the source program have to be converted into local references of the SPMD program. In those cases where the local index set of an array is contiguous, memory allocation is straightforward and the conversion of global references to local references is simple. In case of non-contiguous local index sets, the translation of global to local indices is more complicated and may introduce a large overhead.

In this section we discuss the allocation of distributed data and define functions that convert global indices to local indices and vice versa. We assume that the number of processors is not known at compile time. As a consequence, the size of the local index set of a distributed array must be computed at runtime, and memory must be dynamically allocated.

Assumptions:
Let in the following $A(1:N)$ denote a directly distributed array and $B(1:N)$ an array that is aligned with A. Furthermore, let P denote the number of processors with respect to the corresponding array dimension and let p denote an individual processor with $1 \leq p \leq P$. Let i, $1 \leq i \leq N$, denote a global index and let l, $1 \leq l \leq size(Local(p))$, represent a local index on processor p of A or B respectively. The function $g2l$ transforms from global to local indices and the function $l2g$ transforms local to global indices.

4.2.2.1 Direct Block Distributions

In the case of a $BLOCK$ or $BLOCK(M)$ distribution, the local index set on a particular processor is always contiguous. In order to avoid index conversions at runtime, the local segment of a distributed array will be allocated using global indices.

For sake of consistency, we below list the index conversion functions if memory is allocated based on local indices assuming a lower bound of 1. Index conversion is necessary if a distributed array is statically allocated in the target code[4] because each processor executes the same node program.

$BLOCK$ **Distribution**

$$g2l(i) = i - \mathcal{L}^A(p) + 1 \tag{4.43}$$

$$p = \delta(i) = \left\lceil \frac{i}{M} \right\rceil \tag{4.44}$$

$$l2g(i) = i + \mathcal{L}^A(p) - 1 \tag{4.45}$$

where $\mathcal{L}^A(p)$ denotes the lower bound of the local index set on processor p as defined on page 93.

The size of the local index set on a processor p is given by:

$$\left| Local^A(p) \right| = \begin{cases} \left\lceil \frac{N}{P} \right\rceil & \text{iff } 1 \leq p \leq N \ mod \ P \\ \\ \left\lfloor \frac{N}{P} \right\rfloor & otherwise \end{cases} \tag{4.46}$$

[4]This is only the case for arrays that are not dummy arguments. Dummy arrays can be allocated using global indices by additionally passing the dimension bounds.

$BLOCK(M)$ **Distribution**

If the blocksize M, $M \geq \left\lceil \frac{N}{P} \right\rceil$, is specified explicitly, the formulas 4.43 - 4.45 are still valid but can be formulated independently of the lower bound of the local index set.

$$\begin{aligned} g2l(i) &= 1 + (i-1)\ mod\ M && (4.47) \\ p &= \delta(i) = \left\lceil \frac{i}{M} \right\rceil && (4.48) \\ l2g(l) &= (p-1).M + l && (4.49) \end{aligned}$$

The size of the local index set on a processor p is given by:

$$\left| Local^A(p) \right| = \begin{cases} M & \text{iff } 1 \leq p \leq \left\lfloor \frac{N}{M} \right\rfloor \\ M\ mod\ P & \text{iff } \left\lfloor \frac{N}{M} \right\rfloor \leq p \leq \left\lceil \frac{N}{M} \right\rceil \\ 0 & otherwise \end{cases} \qquad (4.50)$$

4.2.2.2 Implicit Block Distributions

If an array B is aligned with another array A via the alignment function α and A has a $BLOCK[(M)]$ distribution, the index conversion functions are equivalent to those defined in Formula 4.43 and 4.45, but with $\mathcal{L}^A(p)$ replaced by $\mathcal{L}^B_A(p)$ of Formula 4.30 or 4.31.

$$\begin{aligned} g2l(i) &= i - \mathcal{L}^B_A(p) + 1 && (4.51) \\ p &= \delta(\alpha(i)) = \left\lceil \frac{\alpha(i)}{M} \right\rceil && (4.52) \\ l2g(l) &= l + \mathcal{L}^B_A(p) - 1 && (4.53) \end{aligned}$$

4.2.2.3 Direct Cyclic Distributions

If $A(1:N)$ has a $CYCLIC$ distribution, the index conversion functions are given as follows:

$$\begin{aligned} g2l(i) &= \left\lfloor \frac{i-1}{P} \right\rfloor + 1 && (4.54) \\ p &= \delta(i) = 1 + (i-1)\ mod\ P && (4.55) \\ l2g(l) &= (l-1).P + p && (4.56) \end{aligned}$$

The size of the local index set on a processor p is given by:

$$\left| Local^A(p) \right| = max\left(0, \left\lfloor \frac{N-p+P}{P} \right\rfloor\right) \qquad (4.57)$$

4.2.2.4 Implicit Cyclic Distributions

If an array B is aligned with another array A via the alignment function α and A has a $CYCLIC$ distribution, the index conversion functions are given by:

$$g2l(i) = \left\lfloor \frac{(i-1).abs(k)}{lcm(P,k)} \right\rfloor + 1 \quad (4.58)$$

$$p = \delta(\alpha(i)) = 1 + (\alpha(i) - 1) \ mod \ P) \quad (4.59)$$

$$l2g(l) = \left\lfloor \frac{(l-1).lcm(P,k)}{abs(k)} \right\rfloor + \mathcal{L}_A^B(p) \quad (4.60)$$

where k is the stride of the alignment function α, and $\mathcal{L}_A^B(p)$ is the lower bound of $Local_A^B(p)$ as defined in Formula 4.30 and 4.31, respectively.

4.2.2.5 Direct Block-Cyclic Distributions

As already mentioned, if an array $A(1:N)$ has a $CYCLIC(M)$ distribution, the local index set on a particular processor can be described by a union of contiguous index sections. The total number of index sections on a particular processor p is defined by Formula 4.9. The local index set on a particular processor p is given by

σ_1	$(p-1).M+1$	$\ldots$	$p.M$
σ_2	$(p-1).M+P.M+1$	$\ldots$	$p.M+P.M$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_j	$(p-1).M+j.P.M+1$	$\ldots$	$p.M+j.P.M$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

and should be converted to:

σ_1	1	$\ldots$	M
σ_2	$M+1$	$\ldots$	$2.M$
σ_3	$2.M+1$	$\ldots$	$3.M$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_j	$(j-1).M+1$	$\ldots$	$j.M$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Given a global index i, the number of the index section to which i belongs, denoted by $n(i), 1 \leq n(i) \leq n_p$, is given by:

$$n(i) = \left\lfloor \frac{i-1}{M.P} \right\rfloor + 1 \quad (4.61)$$

since from Formula 4.10 we know that the difference of the lower bounds of two consecutive index sections of the local index set is $\mathcal{L}_{i+1}(p) - \mathcal{L}_i(p) = P.M$.

The offset of index i, denoted by $o(i)$, $0 \leq o(i) < M$, in the corresponding index section is given by

$$o(i) = (i-1) \ mod \ M. \tag{4.62}$$

Therefore, the index conversion functions are given by:

$$g2l(i) = o(i) + M.(n(i)-1) + 1 = (i-1) \ mod \ M + M. \left\lfloor \frac{i-1}{M.P} \right\rfloor + 1 \tag{4.63}$$

$$p = 1 + \left(\left\lceil \frac{i}{M} \right\rceil - 1 \right) \ mod \ P \tag{4.64}$$

$$l2g(l) = \left\lfloor \frac{l-1}{M} \right\rfloor .M.P + (p-1).M + 1 + (l-1) \ mod \ M \tag{4.65}$$

4.2.2.6 Implicit Block-Cyclic Distributions

Before we derive the index conversion formula for an array $B(1:N_B)$ that is aligned with an array $A(1:N_A)$ that has a $CYCLIC(M)$ distribution by means of a linear alignment function α we need to take a closer look at the local index set of B. In the following we assume that the local index set is represented according to Formula 4.37 by means of the block representation as a union of contiguous index sections $\bigcup\limits_{j=1}^{n_p} \sigma_j^B(p)$.

Depending on the alignment parameters, some index sections $\sigma_j^B(p)$ at the beginning or at the end may be empty. Let s_p and e_p denote the number of the first and last non-empty index section on a particular processor p, respectively. The number of the index section where the first element of B, $B(1)$, is found is given by $n(\alpha(1))$. Therefore, the number of the first non-empty index section on a particular processor p, s_p, cannot be less than $n(\alpha(1))$. It can be determined by finding the smallest index $i_p, 1 \leq i_p \leq N_B$, such that $\delta(\alpha(i_p)) = p$. This is expressed in the following sketch algorithm:

Algorithm 4.2.1: Computation of first non-empty index section

Input: `array sizes` N_A, N_B
`processor number` p, $1 \leq p \leq P$
`alignment function` $\alpha(i) = k.i + d$
Output: `number of first non-empty index section` s_p
`first index,` i_p`, mapped to processor` p

for j = $n(\alpha(1))$ **to** n_p ! `n() see Formula 4.61`
  if $[\mathcal{L}_j^A(p) : \mathcal{U}_j^A(p) : 1] \cap \alpha([1:N_B]) \neq \phi$ **then** ! `see Formula 4.37`
    $s_p = j$
    $i_p = \alpha^{-1}(max(next(\mathcal{L}_j^A(p), k+d, k), k+d))$
    exit
  end if
end for

From the mathematical point of view, finding the smallest index i of B that is mapped to a particular processor p can be formulated as finding the smallest solution of the set of linear diophantine equations which is constructed as follows:

> Since the lower and upper bound of index section $q+1, 0 \leq q \leq n_p - 1$ of the local index set of the alignment basis on processor p is given by $(p-1).M + q.P.M + 1$ and $p.M + q.P.M$, respectively, we have
>
> $$\begin{array}{lcccl} (p-1).M + q.P.M + 1 & \leq & \alpha(i) & \leq & p.M + q.P.M \\ (p-1).M + 1 & \leq & k.i + d - q.P.M & \leq & p.M \\ (p-1).M - d + 1 & \leq & k.i - q.P.M & \leq & p.M - d \end{array}$$
>
> which corresponds to the set of M linear diophantine equations in the variables i and q given by
>
> $$\{k.i - q.P.M = t \mid (p-1).M - d + 1 \leq t \leq p.M - d\}. \quad (4.66)$$

As a consequence of (4.66), the local index set on a processor p is empty iff $gcd(P.M, k) \not| \, t$, $\forall t$, $(p-1).M - d + 1 \leq t \leq p.M - d$.

Instead of solving the above set of linear diophantine equations we use Algorithm 4.2.1 to compute the smallest index that is mapped to a particular processor. The number of the last non-empty index section, e_p, and the largest index mapped to a processor corresponds to the largest solution in the above set of linear equations. This information is needed to determine the size of the local index set on each processor as described on page 108.

As we will show in the following, it is not necessary to compute all the intersection as defined in Formula 4.37 since the distribution pattern exhibits a periodic behavior. The number of elements in one cycle is dependent on the values of P, M and k. The size of different index sections of $Local^B(p)$, except the first non-empty and the last non-empty index section, differs at most by one.

Theorem 4.2.1:

> *The size of an index section $|\sigma_j^B|$, $s_p < j < e_p$, of the local index set of B is either $\left\lceil \frac{M}{k} \right\rceil$ or $\left\lfloor \frac{M}{k} \right\rfloor$.* ■

Proof:

> *The size of the index sections σ^A of the alignment basis array A is M. The first index i in σ_j^A with which an index of B is aligned has an offset value $o(i)$ with $0 \leq o(i) < k$. If $o(i) = 0$, then we have $|\sigma_j^B| = \left\lceil \frac{M}{k} \right\rceil$. In the case where $o(i) = k-1$ we have $|\sigma_j^B| = \left\lceil \frac{M-k+1}{k} \right\rceil = \left\lfloor \frac{M-k+1+k-1}{k} \right\rfloor = \left\lfloor \frac{M}{k} \right\rfloor$.* ■

Moreover, the lengthes of the index sections of the local index set on a particular processor vary periodically. The length of the cycle is dependent on the parameters M, P, and k only.

Theorem 4.2.2:

The size of consecutive sections of the local index set of an aligned array with a block-cyclic distributed alignment basis varies with a cycle of period length $\mathcal{C} = \left\lfloor \frac{k}{gcd(M.P,k)} \right\rfloor$.
■

Proof:

Let i denote an index of the alignment basis array A with which an index of B is aligned. Index i has an offset value given by $o(i) = (i-1)$ mod M and is located on processor p with $p = 1 + \left\lfloor \frac{(i-1) \ mod \ P.M}{M} \right\rfloor$. We want to find the smallest index i' of A, $i' > i$, on the same processor p with which an index of B is aligned, such that $o(i) = o(i')$. Since the stride of the alignment function is k, $i' = i + q.k$, for some $q > 0$. Let $q' = q.k$. The smallest q' such that

1. $q' = q.k$, *and*
2. q' *mod* $M = 0$, *and*
3. q' *mod* $M.P = 0$

is given by $q' = lcm(M.P, k)$. Since $n(i') - n(i) = \left\lfloor \frac{i + lcm(M.P,k) - 1}{M.P} \right\rfloor - \left\lfloor \frac{i-1}{M.P} \right\rfloor = \left\lfloor \frac{lcm(M.P,k)}{M.P} \right\rfloor = \left\lfloor \frac{k}{gcd(M.P,k)} \right\rfloor$ it holds that $\mathcal{C} = \left\lfloor \frac{k}{gcd(M.P,k)} \right\rfloor$. ■

Condition 2 in the above proof guarantees that index i' has the same offset as i, and condition 3 guarantees that i' is on the same processor as i.

As a direct consequence of Theorem 4.2.2, the local index set $Local_A^B(p)$ of an implicitly distributed block-cyclic array can be constructed by intersection of at most $\mathcal{C} + 1$ index sections of the local index set $Local^A(p)$ of the alignment source array with $\alpha([1 : N_B])$. This is now described more precisely.

Let s_p denote the section number of the first non-empty index section on processor p which is determined according to Algorithm 4.2.1. Furthermore, let $MinLoc_A^B(p)$ denote the first Σ_p elements of $Local_A^B(p)$. $MinLoc_A^B(p)$ is called the **minimal part of the local index set** on processor p. It contains the minimum number of indices that are needed to fully describe the local index set on a particular processor. Note that $MinLoc_A^B(p)$ can be represented by at most $\mathcal{C} + 1$ index sections. It can be constructed as shown in Algorithm 4.2.2.

According to Algorithm 4.2.2, the minimal part of the local index set can be determined by at most $\mathcal{C} + 1$ section intersections. If the offset of the first index on a particular processor is less than the stride of the alignment function, only $\mathcal{C}$ intersections are necessary. Σ_p denotes the number of indices that are contained in one full cycle. From Theorem 4.2.1 it follows that $0 \leq \Sigma_p \leq M$, $\forall p, 1 \leq p \leq P$.

Algorithm 4.2.2: Minimal part of local index set

Input: array sizes N_A, N_B
processor number p, $1 \leq p \leq P$
alignment function $\alpha(i) = k.i + d$
number of first non-empty index section s_p
first index, $i_p \in \sigma_{s_p}(p)$
Output: minimal part of local index set $MinLoc_A^B(p)$
size of $MinLoc_A^B(p)$, Σ_p

```
sec := φ
for j := s_p to min(s_p + C − 1, n_p)          ! n_p see Formula 4.9
    sec := sec ∪ σ_j^B(p)                      ! σ_j^B(p) see Formula 4.37
end for
if offset(α(i_p)) > k then
    lsec := σ_{s_p+C}^B(p)
    for j := 1 to |lsec|
        if offset(α(i_p)) = offset(α(lsec[j])) exit   ! offset see Formula 4.62
        sec = sec ∪ lsec[j]
    end for
end if
MinLoc_A^B(p) := sec
Σ_p := |MinLoc_A^B(p)|
```

Assume that the minimal part of the local index set is given by $MinLoc_A^B(p) = \{l_1, \ldots, l_{\Sigma_p}\}$. The local index set on a particular processor can then be represented alternatively to Formula 4.37 on page 100 by means of the *cyclic representation* as a union of Σ_p non-contiguous index sections as follows:

$$Local_A^B(p) = \bigcup_{j=1}^{\Sigma_p} \hat{\sigma}_j^B(p) = \bigcup_{j=1}^{\Sigma_p} [l\_low_j : l\_up_j : \mathcal{S}_{lis}] \tag{4.67}$$

with $l\_low_j = l\_j$, $l\_up_j = last(l\_low_j, N_B, \mathcal{S}_{lis})$, and the (global) stride $\mathcal{S}_{lis}$ given by

$$\mathcal{S}_{lis} = \left\lfloor \frac{P.M}{gcd(P.M, k)} \right\rfloor \tag{4.68}$$

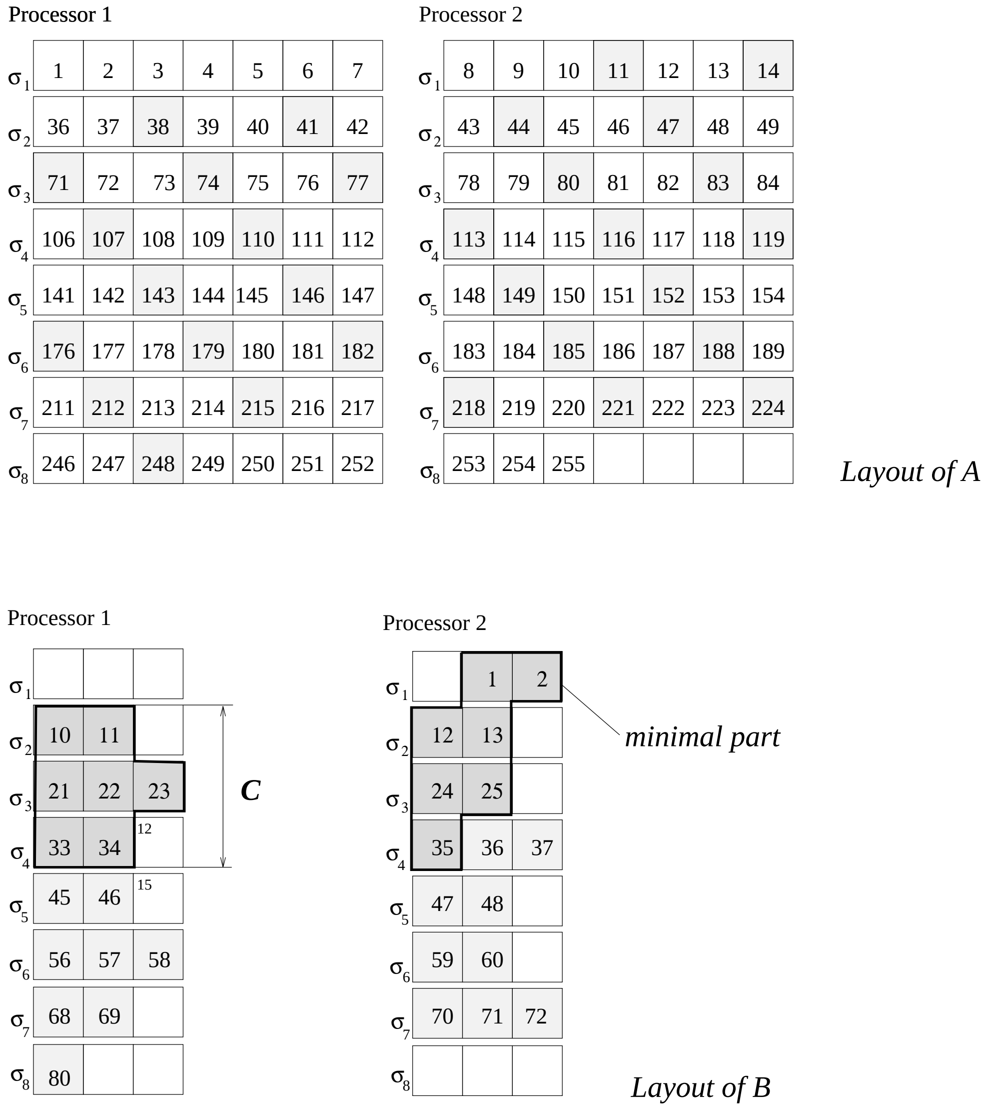
The largest index that is mapped to a particular processor p is denoted by m_p and can be determined as follows:

$$m_p = max_{i=1}^{\Sigma_p}(last(l_i, N_B, \mathcal{S}_{lis})) \tag{4.69}$$

As a consequence, the number of the last non-empty index section $\sigma_{e_p}^B$ on processor p is given by:

$$e_p = n(\alpha(m_p)) \tag{4.70}$$

Example 4.2.1: Minimal part of local index set




```
PROCESSORS R(5)
REAL, DISTRIBUTED(CYCLIC(7))      :: A(255)
REAL, ALIGNED(B(I) WITH A(3*I+8)) :: B(80)
```

Consider the above array declarations. The upper part of the figure shows the local index set of the alignment source array A. The shaded boxes represent indices of A to which an index of B is mapped via the alignment function. The lower part of the figure shows the local index set, and the minimal part of the local index set of array B. The local index set of processor 2 for example can be represented as

$$Local_A^B(p) = [1 :: 35] \cup [2 :: 35] \cup [12 :: 35] \cup [13 :: 35] \cup [24 :: 35] \cup [25 :: 35] \cup [35 :: 35].$$

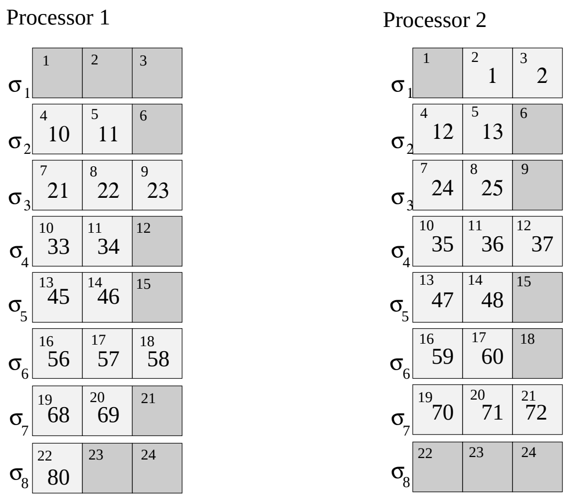
■

4.2.2.6.1 Memory Allocation - Version 1

One possible way of local memory management would be to allocate on a particular processor p an array of size $n_p.\left\lceil \frac{M}{k} \right\rceil$, since the maximum number of elements in one index section according to Theorem 4.2.1 is $\left\lceil \frac{M}{k} \right\rceil$. As a consequence, some memory locations would be unused. The number of the index section where an index i of A is found on some processor p is given by $n(\alpha(i))$. The offset with respect to the index section $\sigma^B_{n(\alpha(i))}$ is given by $\left\lfloor \frac{o(\alpha(i))}{k} \right\rfloor$. A global index i of B would then correspond to the local index l given by:

$$
\begin{aligned}
g2l(i) &= \left\lfloor \frac{o(\alpha(i))}{k} \right\rfloor + \left\lceil \frac{M}{k} \right\rceil .n(\alpha(i)) - 1) + 1 = \\
&= \left\lfloor \frac{(i.k + d - 1)\ mod\ M}{k} \right\rfloor + \left\lceil \frac{M}{k} \right\rceil . \left\lfloor \frac{i.k + d - 1}{M.P} \right\rfloor + 1
\end{aligned}
$$

Example 4.2.2: Memory allocation with gaps




```
PROCESSORS R(5)
REAL, DISTRIBUTED(CYCLIC(7))       :: A(255)
REAL, ALIGNED(B(I) WITH A(3*I+8)) :: B(80)
```

In the above figure the layout of the aligned array B is illustrated. The tiny numbers represent local indices whereas the large numbers represent global indices. The dark-shaded boxes represent unused memory locations. ■

This method is illustrated in Example 4.2.2. The above scheme could be further refined by only taking into account the non-empty index sections σ^B_j, $s_p \leq j \leq e_p$. However, the drawback of this scheme is the waste of memory. Even in the refined version the amount of wasted memory could be up to 50%. For example in the case where $\left\lceil \frac{M}{k} \right\rceil = 1$ and $\left\lfloor \frac{M}{k} \right\rfloor = 0$. Therefore, we apply this index conversion function only in the case where $M > k$ and $k \mid M$.

4.2.2.6.2 Memory Allocation - Minimal Version

In the following we present a memory allocation and index conversion method where on each processor p the exact number of elements of B that are mapped to p are allocated contiguously in order of the global index. The fact that the lengthes of different index sections on a particular processor differ complicates local address calculation. An additional problem is that due to the parameters of the alignment functions the first section might be smaller than $\left\lfloor \frac{M}{k} \right\rfloor$. However, as we have shown before, the lengthes of the index sections of the local index set on a particular processor vary periodically.

We therefore take the approach to determine the period of section lengthes $\mathcal{C}$ as specified above and store on each processor p the accumulated lengthes of the first $\mathcal{C}$ non-empty index sections $\sigma^B_{s_p}(p) \ldots \sigma^B_{sp+\mathcal{C}-1}$ in a table, called **section-period table** and denoted by SP_p. The section-period table on a particular processor p can be determined as follows:[5]

Algorithm 4.2.3: Computation of section-period table

Input: `array sizes` N_A, N_B
`processor number` p, $1 \le p \le P$
`alignment function` $\alpha(i) = k.i + d$
`number of first non-empty index section` s_p
`first index,` i_p`, mapped to processor` p

Output: `section-period table,` SP_p

$SP_p[0] := 0$
for $j := s_p$ **to** $s_p + min(n_p - s_p + 1, \mathcal{C})$
  $SP_p[j - s_p + 1] := SP_p[j - s_p] + |\sigma^B_j(p)|$
end for
$SP_p[1] := SP_p[1] + \left\lfloor \frac{offset(\alpha(i_p))}{k} \right\rfloor$

Since in Algorithm 4.2.3 the offset of the first element i_p is added to $SP_p[1]$, it suffices to store the sizes of at most $\mathcal{C}$ index sections in the table. The section-period table, the number of the first non-empty section, s_p, and the offset of the first element on processor p, denoted by o_p, are needed in the formula for index conversion for arrays with implicit block-cyclic distribution as defined in Theorem 4.2.3.

[5] Of course Algorithm 4.2.2 can be adapted in order to determine SP_p as well.

Theorem 4.2.3:

Given a global index i of array B which has an implicit block-cyclic distribution, the local index l and the corresponding processor number p is given by:

$$g2l(i) = \left\lfloor \frac{n'(i)}{t_p} \right\rfloor .\Sigma_p + SP_p[n'(i) \; mod \; t_p] + o'(i) + 1 \tag{4.71}$$

$$p = \delta(\alpha(i)) = 1 + \left\lfloor \frac{i.k + d - 1}{M} \right\rfloor \; mod \; P \tag{4.72}$$

with

$n'(i) = n(\alpha(i)) - s_p$	...	*relative section number of index i*
$o'(i) = \left\lfloor \frac{o(\alpha(i))}{k} \right\rfloor - o_p$	...	*relative offset of index i*
s_p	...	*the number of the first non-empty section on processor p,* $\lvert\sigma^B_{s_p}(p)\rvert > 0 \;\wedge\; \lvert\sigma^B_j(p)\rvert = 0, \; 1 \le j \le s_p - 1$
o_p	...	*the offset of the first element in index section $\sigma^B_{s_p}(p)$*
$SP_p[0 : t_p]$	...	*the section-period table on processor p* $t_p = min\,(n_p - s_p + 1, \mathcal{C}),\; \Sigma_p = SP_p[t_p]$

■

Proof:

The relative section number $n'(i)$ divided by the table size t_p gives the number of full cycles that index i is away from the first element on processor p. Since Σ_p is the number of elements contained in one cycle, $\left\lfloor \frac{n'(i)}{t_p} \right\rfloor .\Sigma_p$ corresponds to the number of elements contained in full cycles. The elements contained in the remaining $n'(i)$ mod t_p full index sections is given by $SP_p[n'(i) \; mod \; t_p]$. Since the first element in the first non-empty section is not necessarily located in the first position, the offset of index i given by $\left\lfloor \frac{o(\alpha(i))}{k} \right\rfloor$ has to be corrected by subtracting o_p. ■

Note that the SP-table, the offset of the first element o_p and the number of the first non-empty index section have to be computed once for a statically distributed array and can be stored in the array's layout descriptor.

The exact size of the local index set on each processor can be simply determined by applying the index conversion function to the largest index, m_p (Formula 4.69), that has been mapped to a processor.

$$|Local^B_A(p)| = g2l(m_p) \tag{4.73}$$

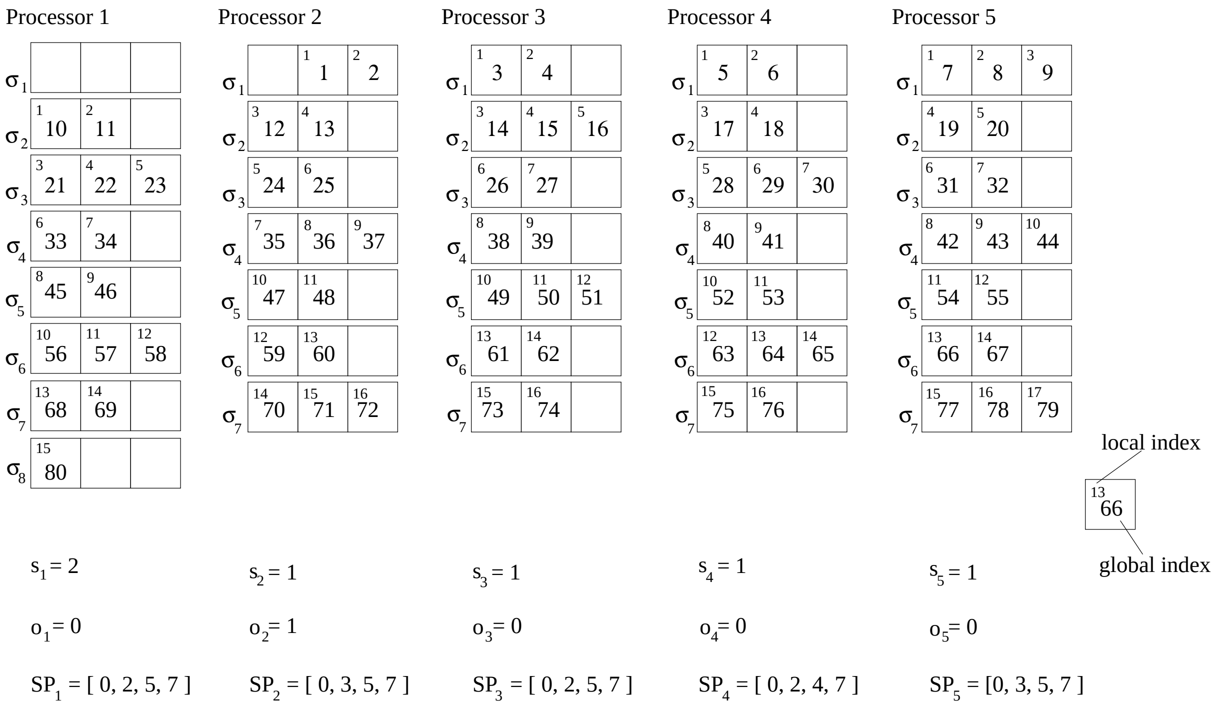


Figure 4.3: Index conversion for implicit block-cyclic distributions

In Figure 4.3 this index conversion method is illustrated for array B of example 4.2.2.

Lemma 4.2.1:

1. *If $k \mid M$, the size of all index sections $s_p < j < e_p$ on a particular processor p is the same.*
2. *If $k \mid M$ and $k \mid P$, the size of all index sections $s_p < j < e_p$ on all processors $p, 1 \leq p \leq P$ is the same.*

■

In all these cases $\mathcal{C} = 1$ and therefore the section period table can be substituted by the constant Σ_p. This can be utilized to simplify the index conversion function 4.71 as follows:

$$g2l(i) \quad = \quad n'(i).\Sigma_p + o'(i) + 1 \tag{4.74}$$

4.2.3 Computation of Work Sets

The next task of the compiler is to determine how to spread the work among the available processors. As already mentioned, work distribution is derived on the basis of the *owner computes* paradigm. This implies that assignment statements (or instances thereof) involving distributed arrays are executed exclusively by those processors which are *owners* of the *lhs* variable. We use the concept of work sets in order to describe work distribution at compile time or/and at runtime.

In the case of array assignments the work set on a particular processor p contains all elements of the *lhs* array section for which p has to execute the assignment. If the *lhs* variable of the assignment statement is a regular array section, the work sets can be represented by either a single or a union of index sections using Fortran 90 triplet notation. In the following we analytically derive closed form expressions for the computation of work sets for array assignments. The presented formulas are based on regular section intersection. For block distributions and cyclic distributions the work set can always be described via a single index section whereas for block-cyclic distributions this is usually not the case. If the *lhs* array section of an assignment statement is irregular (i.e. a vector subscript), the work set cannot be determined by means of section intersection, and cannot be represented efficiently by means of triplets.

Consider an array assignment of the form $A(l : u : s) = ...$; The work set for this assignment statement according to the owner computes rule on a particular processor p is determined by the intersection of the reference space $[l : u : s]$ with the local index set of p.

$$Work^A(p) \quad = \quad Local^A(p) \cap [l : u : s]$$

This intersection can be computed by using the Formulas 4.20 - 4.27. If the local index set of the array on the left hand side of the assignment statement consists of more than one index section, the work set can be determined by the intersection of the *lhs* reference space with every index section of the local index set. In Section 4.2.3.4 we present a method that determines for block-cyclic distributed arrays the minimum number of intersections needed for the computation of work sets.

4.2.3.1 LHS Array with $BLOCK(M)$ Distribution

Given an array $A(1 : N)$ with *BLOCK* or *BLOCK(M)* distribution the work set on a particular processor p with respect to an array section $A(l : u : s)$ is given by:

Case 1: $s > 0$

$$Work^A(p) = [\ max(next(\mathcal{L}^A(p), l, s), l)\ :\ min(u, \mathcal{U}^A(p))\ :\ s] \tag{4.75}$$

Case 2: $s < 0$

$$\begin{aligned} Work^A(p) &= [L(p) : U(p) : s] && (4.76)\\ L(p) &= last(U(p), min(l, \mathcal{U}^A(p)), -s) && (4.77)\\ U(p) &= max(next(\mathcal{L}^A(p), last(l, u, s), -s), last(l, u, s)) && (4.78) \end{aligned}$$

with $\mathcal{L}^A(p)$ and $\mathcal{U}^A(p)$ as defined in Section 4.2.1.1.

As a consequence, an array assignment statement of the above form[6] can be represented in the SPMD program by a single array assignment $A(l' : u' : s') = ...$ where l', u', s' are the lower bound, upper bound and stride of the executing processor. This is also the case for arrays with a cyclic distribution.

4.2.3.2 LHS Array with $CYCLIC$ Distribution

If an array $A(1 : n)$ has a $CYCLIC$ distribution, then the work set according to an array section $A(l : u : s)$ can be represented by means of a single array section given by:

Case 1: $s > 0$

$$Work^A(p) = [first(p, P, l, s) : min(N, u) : lcm(P, s)] \tag{4.79}$$

Case 2: $s < 0$

$$\begin{aligned} Work^A(p) &= [L(p) : U(p) : -lcm(P, s)] && (4.80)\\ L(p) &= last(U(p), min(N, l), lcm(P, s)) && (4.81)\\ U(p) &= first(p, P, last(l, u, s), abs(s)) && (4.82) \end{aligned}$$

4.2.3.3 LHS Array with $CYCLIC(M)$ Distribution

The work set on processor p with respect to an array section $A(l : u : s)$, with array A having a *CYCLIC(M)* distribution, can be computed as follows:

$$Work^A(p) = \bigcup_{i=1}^{n_p}([\mathcal{L}_j^A(p) : \mathcal{U}_j^A(p) : 1] \cap [l : u : s]) \tag{4.83}$$

where $j = i$, for $s > 0$, and $j = n_p - i + 1$ for $s < 0$ and $\mathcal{L}_j^A(p)$ and $\mathcal{U}_j^A(p)$ are the lower and upper bound of the $j - th$ index section as defined in Formula 4.10 and 4.11.

Case 1: $s > 0$

$$Work^A(p) = \bigcup_{i=1}^{n_p}\Big([\ max(next(\mathcal{L}_i^A(p), l, s), l)\ :\ min(u, \mathcal{U}_i^A(p))\ :\ s]\Big) \tag{4.84}$$

Case 2: $s < 0$

$$\begin{aligned} Work^A(p) &= \bigcup_{i=0}^{n_p-1}([L_i(p) : U_i(p) : s]) && (4.85)\\ L_i(p) &= last(U_i(p), min(l, \mathcal{U}_{n_p-i}^A(p)), -s) && (4.86)\\ U_i(p) &= max(next(\mathcal{L}_{n_p-i}^A(p), last(l, u, s), -s), last(l, u, s)) && (4.87) \end{aligned}$$

[6] We assume here that the *rhs* references are local to the executing processor.

Of course it will not always be necessary to compute the intersection of the reference space with all n_p index section. Computation of intersections that will be empty can be avoided by simply comparing the bounds of an index section $\mathcal{L}_j^A$ and $\mathcal{U}_j^A$ with the bounds l and u of the array section. In Formula 4.83 the work set was determined using the block representation of the local index set. Alternatively to this approach, the work set of an array with a block-cyclic distribution can be determined based on the cyclic representation as defined in Formula 4.67. This might be more efficient for small block sizes.

The work set of an implicitly distributed array can be determined in the same way as specified above, by intersection of all non-empty index sections of the local index set with the reference space. However, the local index set of a directly or implicitly distributed block-cyclic array exhibits a periodic behavior, which can be exploited in order to determine the work set in a more efficient way. This method is presented in the following section.

4.2.3.4 Work Distribution Computation for Block-Cyclic Distributions

In this section we show how the work set of a directly or implicitly block-cylic distributed array can be determined with a minimum number of intersections of the index sections of the local index set[7] with the corresponding reference space $[l : u : s]$.

Theorem 4.2.4:

Let B denote an array with an implicit block-cyclic distribution with $\delta^B = \delta^A \circ \alpha_A^B$. Let $i \in Work^B(p)$ denote the first index of the work set on processor p. The next index $i' \in Work^B(p)$ for which $o(\alpha(i)) = o(\alpha(i))$ can be found after exactly $\mathcal{C}_w = \left\lfloor \frac{k.s}{gcd(P.M,k.s)} \right\rfloor$ index sections. ■

Proof:

Let $j \in Local^A(p)$ denote the index of A with which i is aligned. The index $j' \in Local^A(p)$ with which i' is aligned can be represented in the form $j' = j + q.k.s$, for some $q > 0$. Let $q' = q.k.s$. The smallest q' such that

1. $q' = q.k.s$, *and*
2. $q' \bmod M = 0$, *and*
3. $q' \bmod M.P = 0$

is given by $q' = lcm(M.P, k.s)$. Since $n(i') - n(i) = \left\lfloor \frac{i+lcm(M.P,k.s)-1}{M.P} \right\rfloor - \left\lfloor \frac{i-1}{M.P} \right\rfloor = \left\lfloor \frac{lcm(M.P,k.s)}{M.P} \right\rfloor = \left\lfloor \frac{k.s}{gcd(M.P,k.s)} \right\rfloor$ it holds that $\mathcal{C}_w = \left\lfloor \frac{k.s}{gcd(M.P,k.s)} \right\rfloor$. ■

As a consequence, the work set can be constructed by intersecting at most $\mathcal{C}_w + 1$ index sections of the local index set with the reference space. As we have shown before, the local index set of an implicitly block-cyclic distributed array is periodic with cycle $\mathcal{C}$. The number of elements on

[7] If not stated otherwise, we assume that the local index set is represented via the block representation.

processor p in one cycle is given by Σ_p. In order to determine the work set, we have to examine at most $\left\lfloor \frac{\mathcal{C}_w}{\mathcal{C}} \right\rfloor$ full cycles of the local index set of B on processor p. As it will be seen the work set corresponding to an assignment to a regular array section $A(l:u:s)$ can then be represented by means of at most M array sections.

This method is now described more precisely:

Let l_p denote the global index of the first element of the work set on processor p, with $Work(p) \neq \phi$. It can be computed by finding the first non-empty index section of the work set, denoted by ws_p, similarly to Algorithm 4.2.1.

Algorithm 4.2.4: Computation of first element of $B(l:u:s)$ on processor p

Input: array sizes N_A, N_B
processor number p, $1 \leq p \leq P$
alignment function $\alpha(i) = k.i + d$
subscript triplet $[l:u:s]$
Output: first index, $l_p \in [l:u:s]$, mapped to processor p
number of index section, ws_p, containing l_p

for j = $n(\alpha(l))$ **to** n_p ! $n()$ see Formula 4.61
  if $\sigma_j^B(p) \cap [l:u:s] \neq \phi$ **then** ! $\sigma_j^B(p)$ see Formula 4.37
    ws_p = j
    l_p = $max(next(\mathcal{L}_j^B(p), l, s), l)$
    exit
  end if
end for

Let $RelLoc_A^B(p) \subseteq Local_A^B(p)$ denote the set of rl_p consecutive indices of $Local_A^B(p)$ starting with index l_p. $RelLoc_A^B(p)$ is called the **relevant part of the local index set** with respect to the reference space $[l:u:s]$. It contains the minimum number of elements of the local index set that have to be examined in order to determine a processors work set.

$$RelLoc_A^B(p) \subseteq Local_A^B(p), \quad |RelLoc_A^B(p)| = rl_p, \quad rl_p = \Sigma_p. \left\lfloor \frac{\mathcal{C}_w}{\mathcal{C}} \right\rfloor \tag{4.88}$$

Let $MinWork^B(p)$ denote the set of the first $\mathcal{W}_p$ elements ($1 \leq \mathcal{W}_p \leq M$) of the work set on processor p: $MinWork^B(p)$ is called the **minimal part of the work set** on processor p and can be determined by intersection of the relevant part of the local index set $RelLoc_A^B(p)$ with the reference space $[l:u:s]$.

$$MinWork^B(p) = \{w_i \mid w_i \in (RelLoc_A^B(p) \cap [l:u:s]), 0 \leq i \leq \mathcal{W}_p\}. \tag{4.89}$$

$MinWork^B(p)$ can be determined by means of Algorithm 4.2.5. Note that $MinWork^B(p)$ can be represented by at most $\mathcal{C}_w + 1$ index sections with stride s.

Algorithm 4.2.5: Minimal part of work set

Input: `array sizes` N_A, N_B
`processor number` p, $1 \leq p \leq P$
`alignment function` $\alpha(i) = k.i + d$
`number of first non-empty index section` s_p
`first index,` l_p`, mapped to processor` p
`subscript triplet` $[l : u : s]$

Output: `minimal part of work set` $MinWork^B(p)$
`size of` $MinWork^B(p)$, $\mathcal{W}_p$

```
sec := φ
for j := ws_p to min(ws_p + C_w − 1, n_p)          ! n_p see Formula 4.9
    sec := sec ∪ σ_j^B(p) ∩ [l : u : s]
end for
if offset(α(i_p)) > k then
    lsec := σ_j^B(p) ∩ [l : u : s]                  ! σ_j^B(p) see Formula 4.37
    for j := 1 to |lsec|
        if offset(α(l_p)) = offset(α(lsec[j])) exit
        sec = sec ∪ lsec[j]
    end for
end if
MinWork^B(p) := sec
W_p := |MinWork^B(p)|
```

Assume that $MinWork^B(p) = \{w_1, \ldots, w_{\mathcal{W}_p}\}$ is given. The work set on processor p in *global indices* can then be represented by means of the *cyclic representation* as follows:

$$Work^B(p) = \bigcup_{i=1}^{\mathcal{W}_p} [w\_low_i : w\_up_i : \mathcal{S}_{wrk}] \tag{4.90}$$

with $w\_low_i = w_i$, $w\_up_i = last\,(w\_low_i, u, \mathcal{S}_{wrk})$, and the stride $\mathcal{S}_{wrk}$ given by

$$\mathcal{S}_{wrk} = \left\lfloor \frac{s.P.M}{gcd(P.M, k.s)} \right\rfloor \tag{4.91}$$

The largest index of the work index set on a particular processor p, denoted by mw_p, can be determined as follows:

$$wm_p = max_{i=1}^{\mathcal{W}_p}(last(w\_low_i, N_B, \mathcal{S}_{wrk}))) \tag{4.92}$$

The number of the last non-empty index section of the work index set (with respect to the block representation) on a processor p is given by:

$$we_p = n(\alpha(wm_p)) \tag{4.93}$$

By using the function $g2l()$, Σ_p and the SP table from Formula 4.71 we immediately can construct the work set with respect to *local indices*, denoted by $work^B(p)$, as follows:

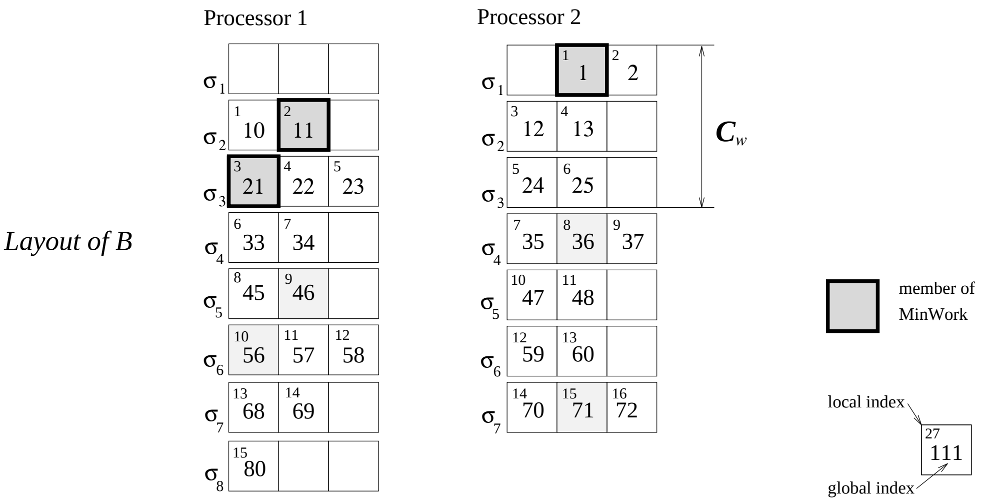
$$work^B(p) \quad = \quad \bigcup_{i=1}^{\mathcal{W}_p} [g2l(w\_low_i) : g2l(w\_up_i) : loc\_\mathcal{S}_{wrk}] \tag{4.94}$$

with

$$loc\_\mathcal{S}_{wrk} = \Sigma_p . \left\lfloor \frac{\mathcal{C}_w}{\mathcal{C}} \right\rfloor = \Sigma_p . \left\lfloor \frac{s.gcd(P.M, k)}{gcd(P.M, k.s)} \right\rfloor \tag{4.95}$$

Note that the stride of the work set with respect to local indices is dependent on Σ_p and thus may be different for different processors p, $1 \leq p \leq P$.

Example 4.2.3: Work set of a regular array section




```
PROCESSORS R(5)
REAL, DISTRIBUTED(CYCLIC(7))      :: A(250)
REAL, ALIGNED(B(I) WITH A(3*I+8)) :: B(80)

    B(1::5) = ...
```

As can be seen from the figure, the work set for the array section $B(1::5)$ *for processor 1 is given in global indices by* $[11::35] \cup [21::35]$ *and in local indices by* $[2::7] \cup [3::7]$. *For processor 2 the work set consists of only 1 section: in global indices* $[1::35]$ *and in local indices* $[1::7]$. ■

In Example 4.2.3 we illustrate the work sets on the first two processors for an assignment to the array section $B(::5)$.

Under the assumption that the right hand side expression is replicated, an array assignment $B(l : u : s) = ...$ can be transformed into SPMD code as shown in Example 4.2.4 (Method 1). Since the size of the minimal part of the work set is bound by M, a single array assignment statement is split into at most M array assignment statements.

Example 4.2.4: SPMD code for $B(l : u : s) = ...$

Method 1:

```
I=1,SIZE(minWork)
  B(minWork(I) : last : s_wrk) = ...
END DO
```

Method 2:

```
base = minWork(1); I = 0;
period = SIZE(minWork);
DO WHILE (base <= last)
   A(base) = ...
   base = base + TABLE(I)
   I = MODULO((I+1),period)
END DO
```

∎

Another method of code generation for array assignments was suggested by Chatterjee et.al. [29]. They represent the sequence of local memory accesses for an array section by means of a finite state machine with at most M cycles. This information is stored in a table and the assignment statement is transformed in a while-loop as shown in the right part of Example 4.2.4. In each iteration of the loop a single array element is updated and the offset of the next element that has to be updated is determined from the table. Whereas this method might result in a better cache utilization, method 1 attempts to retain the structure of the array assignment. As a consequence, method 1 can exploit not only inter-processor parallelism but also intra-processor parallelism since for each array assignment statement the vector-units (if any) of the executing processor can be utilized.

The sequence of local memory accesses can be easily determined be scanning once through the minimal part of the work set as shown in the following sketch algorithm.

Algorithm 4.2.6: Construction of memory access sequence for array sections

Input: minimal work set $MinWork^B(p) = \{w_1, \ldots, w_{\mathcal{W}_p}\}$
$\Sigma_p,\ \mathcal{C}_w,\ \mathcal{C},\ \mathcal{W}_p$
Output: memory access sequence table $\mathcal{M}_p$ for processor $p, 1 \le p \le P$

$\mathcal{M}_p = 0$
for $i = 1, \mathcal{W}_p - 1$
  $\mathcal{M}[i] = g2l(w_{i+1}) - g2l(w_i)$
end for
$\mathcal{M}[\mathcal{W}_p] = g2l(w_1) + \Sigma_p . \left\lfloor \frac{\mathcal{C}_w}{\mathcal{C}} \right\rfloor - g2l(w_{\mathcal{W}_p})$

For the assignment statement of Example 4.2.3 the memory access sequence table on processor 1 is given by $\mathcal{M}_1 = \{1, 6\}$ and on processor 2 by $\mathcal{M}_2 = \{7\}$.

4.2.3.5 DO Loops, FORALL Statements

The iteration space defined by nested DO loops in general needs not to be rectilinear (e.g. when the loop bounds of the inner loops of a loop nest depend on functions of outer loop indices). Furthermore, subscripts for array elements referenced in the loop body are not restricted to linear functions. As a consequence, the formulas for the computation of work sets as derived in Section 4.2.3 may be applied to DO loops only if all of the following conditions are satisfied:

- the iteration space is rectilinear
- all array subscripts are linear functions of the form $k.i + d$, where i is a loop iteration variable, and k and d are integer expressions which must not depend on any loop iteration variable
- no loop iteration variable must appear more than once in a reference to a distributed array (no interaction of dimensions)
- no iteration affects any other iteration directly or indirectly, (no loop carried dependencies)

If the above restrictions are not satisfied, work sets must be represented on an element basis rather than via regular sections.

4.2.4 Computation of Communication Sets

For the parallelization of assignment statements that reference distributed arrays each processor has to determine the indices of non-local elements it must receive from other processors in order to perform all the computations defined by its work set. Furthermore, it also has to determine which local elements have to be sent to other processors. Communication sets determine those elements a particular processor p must send (or receive) to (from) another processor q, in order to perform the computations that are defined by its work set.

In case of array assignment statements, involving regular array sections, computation of communication sets is based on regular section intersection as described in Section 4.2.1.2.1. For DO loops or FORALL statements send sets and receive sets usually have to be determined on a single element basis, unless the restrictions stated in Section 4.2.3.5 are satisfied. In the following we present a general method for the computation of communication sets for (regular) array assignments. The goal of this approach is to avoid (whenever possible) single element messages. Therefore, similar to local index sets and work sets, communication sets will be represented as regular sections or unions thereof.

In order to compute the send index sets $Send^B(p,q)$ and receive index sets $Recv^B(p,q)$ as defined in Section 3.4, $\forall q \in P, q \neq p$, on a particular processor p for an array assignment statement of the form $A(l:u:s) = B(l':u':s')$, usually the following actions have to be taken.

for each processor q, $q \neq p$

1. Generate the work set for processor q, $Work^A(q)$.
2. Determine the linear function that maps elements of the left hand side to the corresponding elements on the right hand side. This function, in the following denoted by f, is given by:

$$f(i) = \left\lfloor \frac{(i-l).s'}{s} \right\rfloor + l' \tag{4.96}$$

3. Determine the transformed work set $f(Work^A(q))$.
4. Compute the intersection $f(Work^A(q)) \cap Local^B(p)$ which corresponds to $Send^B(p,q)$.

Note, since $Send(p,q)^B = Recv^B(q,p)$ the receive set is implicitly determined by the send set.

If both the *lhs* array and the *rhs* array have a block or cyclic distribution, then the send index set of a processor p with respect to a processor q can always be represented by means of a single section. This might not be the case if one of the arrays involved in an array assignment has a block-cyclic distribution. In this case the communication sets have to be determined by intersection of all index sections of the work set of the left hand side array with all index sections of the local index set of the right hand side array. As a consequence, also the communication sets have to be represented by a union of index sections. In order to reduce the number of message transfers that are necessary for the execution of a parallelized assignment statement, it is necessary to pack the sub-messages[8] between two particular processors into a buffer. This allows sending all data that have to be exchanged between two processors in a single message. In order to guarantee correctness, the sending and receiving processor must use the same mechanisms for packing and unpacking.

In the following we first consider the most general case where both arrays A and B have an implicit block-cyclic distribution. Depending on whether the work set for A and the local index set of B are represented via the block-representation or the cyclic-representation, there are four different ways to determine the communication sets for the above assignment statement.

4.2.4.1 Cyclic-Based Communication Set Generation

In the first method for generation of communication sets we assume that for both the local index set and the work set the cyclic representation is used. In this case the elements of B a processor p has to send to a processor q can be determined by intersection of the work set on processor q (as defined in Formula 4.90) on which the mapping function f has been applied with the local index set of processor p (as defined in Formula 4.67). The formulas presented below are based on the Vienna Fortran 90 program in Example 4.2.5.

Example 4.2.5: Computation of Communication Sets

```
PROCESSORS R(PP)
REAL, DISTRIBUTED(CYCLIC(M))  TO R(1:P)  :: X(NX)
REAL, DISTRIBUTED(CYCLIC(M')) TO R(1:P') :: Y(NY)
REAL, ALIGNED(A(I) WITH X(K*I+D)  )      :: A(NA)
REAL, ALIGNED(B(I) WITH X(K'*I+D'))      :: B(NB)

   A(l:u:s) = B(l':u':s')
```

■

Computation of Send Sets

The *cyclic representation* of the send index set $Send^B(p,q)$ for the array assignment statement of Example 4.2.5 can be determined as follows:

$$\begin{aligned} Send^B(p,q) &= f(Work^A(q)) \cap Local^B(p) = \\ &= f(\bigcup_{i=1}^{\mathcal{W}_q^A} \left[w\_low_i : w\_up_i : \mathcal{S}_{wrk}^A\right]) \cap (\bigcup_{j=1}^{\Sigma_p^B} \left[l\_low_j : l\_up_j : \mathcal{S}_{lis}^B\right]) = \\ &= (\bigcup_{i=1}^{\mathcal{W}_q^A} \left[f(w\_low_i) : f(w\_up_i) : \hat{\mathcal{S}}_{wrk}^A\right]) \cap (\bigcup_{j=1}^{\Sigma_p^B} \left[l\_low_j : l\_up_j : \mathcal{S}_{lis}^B\right] = \\ &= \bigcup_{k=1}^{\mathcal{T}_{qp}} \left[s\_low_k : s\_up_k : \mathcal{S}_{snd}^B\right] \end{aligned} \tag{4.97}$$

[8] A sub-message corresponds to elements described by one index section.

with $\hat{\mathcal{S}}^A_{wrk} = \left\lfloor \frac{\mathcal{S}^A_{wrk}.s'}{s} \right\rfloor$, $0 \leq \mathcal{T}_{qp} \leq \mathcal{W}^A_q.\Sigma^B_p$, and the (global) stride of the send index set $\mathcal{S}^B_{snd}$ is given by

$$\begin{aligned} \mathcal{S}^B_{snd} &= lcm(\hat{\mathcal{S}}^A_{wrk}, \mathcal{S}^B_{lis}) = \\ &= lcm\left(\left\lfloor \frac{s'.P.M}{gcd(P.M, k.s)} \right\rfloor, \left\lfloor \frac{P'.M'}{gcd(P'.M', k')} \right\rfloor\right) = \\ &= \left\lfloor \frac{s'.P.M.P'.M'}{\mathcal{G}} \right\rfloor \end{aligned} \tag{4.98}$$

and

$$\begin{aligned} \mathcal{G} &= gcd(P.M, k.s).gcd(P'.M', k').gcd\left(\frac{s'.P.M}{gcd(P.M, k.s)}, \frac{P'.M'}{gcd(P'.M', k')}\right) = \\ &= gcd\left(s'.P.M.gcd(P'.M', k'), P'.M'.gcd(P.M, k.s)\right) \end{aligned} \tag{4.99}$$

Now, by applying the function for conversion of global indices to local indices, denoted by $g2l^B$, as defined in Formula 4.71, the send index set in local indices, denoted by $send^B(p,q)$, is given by:

$$send^B(p,q) = \bigcup_{k=1}^{\mathcal{T}_{qp}} \left[g2l^B(s\_low_k) : g2l^B(s\_up_k) : loc\_\mathcal{S}^B_{snd}\right] \tag{4.100}$$

with the local stride of the send index set $loc\_\mathcal{S}^B_{snd}$ given by

$$loc\_\mathcal{S}^B_{snd} = \left\lfloor \frac{s'.P.M.\Sigma^B_p.gcd(P'.M', k')}{\mathcal{G}} \right\rfloor \tag{4.101}$$

Computation of Receive Sets

The receive index set (in global indices) can be computed in the same way as shown in the previous section for the send index set. Since $Recv^B(p,q) = Send^B(q,p)$, the receive index sets can be determined from the send index sets (and vice versa). For sake of consistency, we give below the formula for computation of $Recv^B(p,q)$.

$$\begin{aligned} Recv^B(p,q) &= f(Work^A(p)) \cap Local^B(q) = \\ &= f(\bigcup_{i=1}^{\mathcal{W}^A_p} \left[w\_low_i : w\_up_i : \mathcal{S}^A_{wrk}\right]) \cap (\bigcup_{j=1}^{\Sigma^B_q} \left[l\_low_j : l\_up_j : \mathcal{S}^B_{lis}\right]) = \\ &= \bigcup_{i=1}^{\mathcal{W}^A_p} \left[f(w\_low_i) : f(w\_up_i) : \hat{\mathcal{S}}^A_{wrk}\right] \cap (\bigcup_{j=1}^{\Sigma^B_q} \left[l\_low_j : l\_up_j : \mathcal{S}^B_{lis}\right]) = \\ &= \bigcup_{k=1}^{\mathcal{T}_{pq}} \left[r\_low_k : r\_up_k : \mathcal{S}^B_{snd}\right] \end{aligned} \tag{4.102}$$

with $0 \leq \mathcal{T}_{pq} \leq \mathcal{W}_p^A.\Sigma_q^B$, and $\hat{\mathcal{S}}_{wrk}^A$ and $\mathcal{S}_{snd}^B$ as defined in Formula 4.98.

The receive set as defined above specifies which elements of the right hand side array B a processor has to receive. In order to determine the corresponding indices of elements of the left hand side array A, the inverse of function f has to be applied.

$$Recv^A(p,q) = f^{-1}(Recv^B(p,q)) = \bigcup_{k=1}^{\mathcal{T}_{pq}} \left[r\_low_k : r\_up_k : \mathcal{S}_{rcv}^A\right] \tag{4.103}$$

with $r\_low_k = f^{-1}(s\_low_k)$ and $r\_up_k = f^{-1}(s\_up_k)$ and the global stride of $Recv^A(p,q)$, denoted by $\mathcal{S}_{rcv}^A$ given by

$$\mathcal{S}_{rcv}^A = lcm(\mathcal{S}_{wrk}^A, \mathcal{S}_{lis}^B) = \left\lfloor \frac{s.P.M.P'.M'}{\mathcal{G}} \right\rfloor \tag{4.104}$$

$Recv^A(p,q)$ is called the receive index set of p from q with respect to A. The global to local index conversion of the send index set as described above has been realized by applying the index conversion function with respect to the right hand side array B. However, for global to local conversion of the set $Recv^A(p,q)$ we have to apply the index conversion with respect to the left hand side array A, denoted by $g2l^A$. The receive index set in local indices, denoted by $recv^A(p,q)$, therefore is given by:

$$recv^A(p,q) = \bigcup_{k=1}^{\mathcal{T}_{pq}} \left[g2l^A(r\_low_k) : g2l^A(r\_up_k) : loc\_\mathcal{S}_{rcv}^A\right] \tag{4.105}$$

with the local stride of the receive set $loc\_\mathcal{S}_{rcv}^A$ given by

$$loc\_\mathcal{S}_{rcv}^A = \left\lfloor \frac{s.P'.M'.\Sigma_q^A.gcd(P.M,k)}{\mathcal{G}} \right\rfloor \tag{4.106}$$

If the arrays involved in an assignment statement have a block-cyclic distribution with large block sizes, then the above approach might be very inefficient since the send and receive set may consist of a very large number of index sections. Furthermore, the values of the strides of the communication set may be very large resulting in a bad utilization of the cache. In such cases communication sets can be generated more efficiently if local index sets and work sets are based on the block representation. This is described in the next section.

4.2.4.2 Block-Based Communication Set Generation

Assume that for the assignment statement in Example 4.2.5 the work set with respect to the left hand side array section and the local index set of the right hand side array are represented by means of the block representation[9] as follows:

[9]Note that in this case the local index set consists of a union of contiguous index sections (i.e. with stride = 1) whereas the stride of the index sections of the work index set is s.

$$Work^A(q) = \bigcup_{i=ws_q}^{we_q} \sigma_i^A \quad (4.107)$$

$$Local^B(p) = \bigcup_{j=s_p}^{e_p} \sigma_j^B \quad (4.108)$$

where s_p and e_p are the numbers of the first and last non-empty index section of the local index set of B on processor p and ws_q and we_q are the number of the first and last non-empty index section of the work set of processor q. The *block representation* of the communication sets therefore can be determined by $(we_q - ws_q + 1).(e_p - s_p + 1)$ section intersections. Depending on the relationship between the distribution of A and B and the access pattern, many of these intersections may be empty.

However, since both the local index set and the work index set exhibit a periodic behavior, not all these intersections have to be computed explicitly. We will show that the number of index sections of the work set that have to be examined until the communication pattern repeats is given by $\mathcal{C}_{sw}$. Similarly, the number of index sections of the local index set of the *rhs* array that have to be examined until the communication pattern repeats is denoted by $\mathcal{C}_{sl}$.

Therefore, instead of intersecting all index sections of the work index set with all index sections of the local index set it is sufficient to determine only the intersection of $\mathcal{C}_{sw}$ index sections of the work index set with $\mathcal{C}_{sl}$ index sections of the local index set. The quantities $\mathcal{C}_{sw}$ and $\mathcal{C}_{sl}$ can be determined as follows:

Theorem 4.2.5:

Consider an assignment statement $A(l : u : s) = B(l' : u' : s')$ where both arrays A and B have an implicit block-cyclic distributions as shown in Example 4.2.5. Let $i \in Send^B(p, q)$ denote the first index of the send index set of processor p with respect to processor q and let the global stride of the send index set be given by $\mathcal{S}_{snd}^B$. Furthermore, let $j \in Work^A(p)$, $j = f^{-1}(i)$, denote the index of the work set of processor q that corresponds to index i.

1. *The next index $j' \in Work^A(p)$ for which $f(j') = f(i + \mathcal{S}_{snd}^B)$ is satisfied can be found after exactly $\mathcal{C}_{sw} = \left\lfloor \frac{k.s.P'.M'}{\mathcal{G}} \right\rfloor$ index sections.*

2. *Similarly, the next index $i' \in Local^B(q)$ for which $i' = i + \mathcal{S}_{snd}^B$ can be found after exactly $\mathcal{C}_{sl} = \left\lfloor \frac{k'.s'.P.M}{\mathcal{G}} \right\rfloor$ index sections.*

According to Formula 4.99 $\mathcal{G} = gcd\,(s'.P.M.gcd(P'.M', k'), P'.M'.gcd(P.M, k.s))$.

■

Proof:

1. *The cycle of the work index set* $Work^A(q)$ *according to Theorem 4.2.4 is* $\mathcal{C}_w = \left\lfloor \frac{k.s}{gcd(P.M,k.s)} \right\rfloor$ *and the global stride of the work index set according to Formula 4.91 is given by* $\mathcal{S}^A_{wrk} = \left\lfloor \frac{s.P.M}{gcd(P.M,k.s)} \right\rfloor$. *The global stride of the send index set with respect to the lhs array can be derived from Formula 4.98 as* $\hat{\mathcal{S}}^B_{snd} = \left\lfloor \frac{\mathcal{S}^B_{snd}.s}{s'} \right\rfloor = \left\lfloor \frac{s.P.M.P'.M'}{\mathcal{G}} \right\rfloor$.
   Using these quantities the cycle of the send index set with respect to the lhs array A can be determined by $\mathcal{C}_{sw} = \left\lfloor \frac{\hat{\mathcal{S}}^B_{snd}}{\mathcal{S}^A_{wrk}} \right\rfloor . \mathcal{C}_w = \left\lfloor \frac{s.P.M.P'.M'}{\mathcal{G}} . \frac{gcd(P.M,k.s)}{s.P.M} . \frac{k.s}{gcd(P.M,k.s)} \right\rfloor = \left\lfloor \frac{k.s.P'.M'}{\mathcal{G}} \right\rfloor$.

2. *The cycle of the local index set* $Local^B(p)$ *according to Theorem 4.2.2 is given by* $\mathcal{C} = \left\lfloor \frac{k}{gcd(P'.M',k')} \right\rfloor$ *and the global stride of the local index set by* $\mathcal{S}^B_{lis} = \left\lfloor \frac{P'.M'}{gcd(P'.M',k')} \right\rfloor$.
   The cycle of the send index set with respect to the local index set of the rhs array can be determined by $\mathcal{C}_{sl} = \left\lfloor \frac{\mathcal{S}^B_{snd}}{\mathcal{S}^B_{lis}} \right\rfloor . \mathcal{C} = \left\lfloor \frac{s'.P.M.P'.M'}{\mathcal{G}} . \frac{gcd(P'.M',k')}{P'.M'} . \frac{k'}{gcd(P'.M',k')} \right\rfloor = \left\lfloor \frac{k'.s'.P.M}{\mathcal{G}} \right\rfloor$.

■

Lemma 4.2.2:

Consider an array assignment statement $A(l : u : s) = B(l' : u' : s')$ *as shown in Example 4.2.5. If* $P = P'$ *and* $M = M'$ *is satisfied, then* $\mathcal{C}_{sw} \leq k.s$ *and* $\mathcal{C}_{sl} \leq k'.s'$. ■

Proof:

From Theorem 4.2.5 we know that $\mathcal{C}_{sw} = \left\lfloor \frac{k.s.P'.M'}{\mathcal{G}} \right\rfloor$. *Since* $P = P'$ *and* $M = M'$ *we have* $\mathcal{C}_{sw} = \left\lfloor \frac{k.s.P.M}{gcd(s'.P.M.gcd(P.M,k'),P.M.gcd(P.M,k.s))} \right\rfloor = \left\lfloor \frac{k.s}{gcd(s'.gcd(P.M,k'),gcd(P.M,k.s))} \right\rfloor$ *and thus* $\mathcal{C}_{sw} \leq k.s$.

$\mathcal{C}_{sl} \leq k'.s'$ *can be shown similarly.*
■

As shown above, the send index set $Send^B(p,q)$ can be determined by intersection of $\left\lfloor \frac{\hat{\mathcal{S}}^B_{snd}}{\mathcal{S}^A_{wrk}} \right\rfloor$ full cycles of the work index set $Work^A(q)$ of processor q with $\left\lfloor \frac{\mathcal{S}^B_{snd}}{\mathcal{S}^A_{lis}} \right\rfloor$ full cycles of the local index set $Local^B(p)$ on processor p.

This is now described more precisely:

Let $MinSend^B(p,q)$ denote the **minimal part of the send index set** $Send^B(p,q)$. It is given by:

$$MinSend^B(p,q) = f(RelWork^A(q)) \cap RelLocS^B(p) \tag{4.109}$$

The **relevant part of the work index set**, $RelWork^A(q)$, contains the minimum number of elements of the work index set $Work^A(q)$ that have to be considered when computing a processors send index set. It is given by the first rw_q indices of the work index set.

$$RelWork^A(q) \subseteq Work^A(q), \quad |RelWork^A(q)| = rw_q, \quad rw_q = \mathcal{W}_q^A \cdot \left\lfloor \frac{\mathcal{C}_{sw}}{\mathcal{C}_w} \right\rfloor \tag{4.110}$$

where $\mathcal{W}_q^A$ is the size of $MinWork^A(q)$. Note that $RelWork^A(q)$ corresponds to the first $\left\lfloor \frac{\hat{\mathcal{S}}_{snd}^B}{\mathcal{S}_{wrk}^A} \right\rfloor$ full cycles of the work index set and thus can be represented by at most $nw_q + 1$ index sections of the work index set, with nw_q given by:

$$nw_q \quad = \quad min\,(we_q - ws_q + 1, C_{sw}) \tag{4.111}$$

where C_{sw} is the cycle of the send index set with respect to $A(l:u:s)$ as defined in Theorem 4.2.5.

The **relevant part of the local index set with respect to the send index set**, denoted by $RelLocS^B(p)$, is defined similarly to Formula 4.88. It is given by rl_p consecutive indices of $Local^B(p)$ starting with the first index of $B(l':u':s')$ that is located on processor p as follows:

$$RelLocS^B(p) \subseteq Local^B(p), \; |RelLocS^B(p)| = rl_p, \; rl_p = \Sigma_p^B \cdot \left\lfloor \frac{\mathcal{C}_{sl}}{\mathcal{C}} \right\rfloor \tag{4.112}$$

where $\mathcal{C}_{sl}$ is the cycle of the send index set with respect to $Local^B(p)$ as defined in Theorem 4.2.5, $\mathcal{C}$ is the cycle of $Local^B(p)$, and Σ_p^B is the number of indices contained in one cycle of the local index set. Note that $RelLocS^B(p)$ can be represented by at most $nl_p + 1$ index sections of the local index set, where nl_p is given by

$$nl_p \quad = \quad min\,(e_p - s_p + 1, \mathcal{C}_{sl}) \tag{4.113}$$

Assume that $MinSend^B(p,q) = \bigcup_{k=1}^{T_{pq}} [low_k : up_k : s']$ is given. The block representation of the send index set $Send^B(p,q)$ then can be determined by adding multiples of $\mathcal{S}_{snd}^B$ to the index sections of $MinSend^B(p,q)$ as shown in Algorithm 4.2.7.

Algorithm 4.2.7: Construction of $Send^B(p,q)$ based on $MinSend^B(p,q)$

Input: minimal part of send index set, $MinSend^B(p,q)$ (Formula 4.109)
index of last element of $Work^A(p)$, wm_p (Formula 4.92)
global stride of send index set, $\mathcal{S}^B_{snd}$ (Formula 4.98)
Output: send index set $Send^B(p,q)$

$Send^B(p,q)$:= $MinSend^B(p,q)$; c := 0
while
  c := $c+1$
  for $k = 1, T_{pq}$
    if $low_k + c * \mathcal{S}_{snd} > f(wm_p)$ **exit while**
    $Send^B(p,q)$:= $Send^B(p,q) \cup [Min(m_p, low_k + c * \mathcal{S}_{snd}) : Min(m_p, up_k + c * \mathcal{S}_{snd}) : s']$
  end for
end while

Since the stride of all the index sections of the transformed work set (i.e. on which the function f has been applied) is s' and all index sections of the local index set are contiguous (i.e have stride 1), the stride of the index sections of the send set in global indices is s' and the stride of the index sections of the receive set (with respect to A) is s. Furthermore, the stride of the index sections in local indices is the same as the global stride.

In Example 4.2.6 the naive approach of communication generation (i.e. the intersection of all index sections of the work index set with all index sections of the local index set) is compared to the optimized method based on the minimal part of the send index set.

4.2.4.3 Comparison of Block-based and Cyclic-based Communication Generation

The preprocessing overhead for the different methods of communication generation depends largely on the number of index sections needed for the representation of the local index sets and the work index sets of the *lhs* and *rhs* arrays. The communication set generation algorithms have been implemented in a pre-prototype runtime support system written in Fortran 90. The function for determining the send index set takes as arguments distribution descriptors of both the *lhs* and *rhs* arrays and the corresponding section subscripts and returns the send index set represented as an array of triplets.

Example 4.2.6 compares the three different methods with respect to the illustrated array assignment statement. All algorithms have been compiled with the NAG Fortran 90 compiler and have been measured on a SUN Sparcstation 2. Since the NAG Fortran 90 compiler is a preprocessor that generates C code, the absolute numbers may be misleading. However, the measurements clearly show the tradeoff between the block-based and cyclic-based methods with respect to varying block sizes. If M is small, then usually the cyclic-based method of communication generation will be more efficient, but may become extremely inefficient for large block sizes. The opposite is true for the block-based method of communication set generation.

Example 4.2.6: Preprocessing overhead of communication set generation algorithms

```
PROCESSORS :: R( 16 )
PARAMETER (N=65536)

REAL, DISTRIBUTED(CYCLIC(M)) :: X(k*N), Y(k'*N)

REAL, ALIGNED(A(I) WITH X(k*I)) :: A(N)
REAL, ALIGNED(B(I) WITH Y(k'*I)) :: B(N)

A(1:39322:3) = ... B(1:65536:5) ...
```

The preprocessing overhead for the computation of the communication set for the above array assignment statement is shown in Figure 4.4. For the case where $k = 2$ and $k' = 3$ and $M = 64$ we have the following situation:

The cycle of the local index set $Local^B$ is given by $\mathcal{C} = \left\lfloor \frac{3}{gcd(16*64,3)} \right\rfloor = 3$ and the cycle of the work index set $Work^A$ is given by $\mathcal{C}_w = \left\lfloor \frac{2*3}{gcd(16*64,2*3)} \right\rfloor = 3$. As a consequence, both the minimal part of the local index set $MinLoc^B(p)$ and the minimal part of the work index set $MinWork^A(q)$ can be represented by means of 3 index sections.

- *Method B naive:*
  The maximum number of index section of the block representation of the local index set $Local^B$ is given by $\left\lfloor \frac{3*N}{P.M} \right\rfloor = \left\lfloor \frac{3*65536}{16*64} \right\rfloor = 192$. The maximum number of index sections of the block representation of the work index set $Work^A$ is given by $\left\lfloor \frac{2*39322}{16*64} \right\rfloor = 77$. As a consequence, $192 * 77 = 14784$ section intersections have to be computed to determine the send index set for a particular pair of processors.

- *Method B opt.:*
  Since $gcd(P.M, k.s) = gcd(16 * 64, 2 * 3) = 2$, $gcd(P.M, k') = gcd(16 * 64, 3) = 1$, and thus $\mathcal{G} = gcd(s'.P.M.gcd(P'.M', k'), P'.M'.gcd(P.M, k.s)) = 1024$. Thus, according to Theorem 4.2.5 the cycle of the send index set with respect to the work index set is given by $\mathcal{C}_{sw} = \left\lfloor \frac{k.s.P'.M'}{\mathcal{G}} \right\rfloor = \left\lfloor \frac{2*3*16*64}{1024} \right\rfloor = 6$. Similarly the cycle of the send index set with respect to the local index set is given by $\mathcal{C}_{sl} = \left\lfloor \frac{k'.s'.P.M}{\mathcal{G}} \right\rfloor = \left\lfloor \frac{3*5*16*64}{1024} \right\rfloor = 15$. As a consequence, the send index set can be determined by only $6 * 15 = 90$ section intersections as opposed to 14784 with the naive method.

- *Method C:*
  The number of index sections for the cyclic representation of the work index set is given by $\mathcal{W}^A = 32$ and for the cyclic representation of the local index set by $\Sigma^B = 64$. As a consequence, $32 * 64 = 2048$ section intersections are needed for the computation of the send index set. The global stride of the work index set is given by $\mathcal{S}^A_{wrk} = 1536$, the global stride of the local index set is given by $\mathcal{S}^B_{lis} = 1024$, and the global stride of the send index set $Send^B$ is given by $\mathcal{S}^B_{snd} = 5120$.

■

M	Method B naive (ms)		Method B opt. (ms)				Method C (ms)	
	$k = 2,\ k' = 3$	$k = k' = 2$	$k = 2,\ k' = 3$		$k = k' = 2$		$k = 2,\ k' = 3$	$k = k' = 2$
			phase 1	total	phase 1	total		
4	23883.5	18758.4	0.79	4.75	0.22	8.39	**0.17**	**0.13**
8	8954.7	7031.3	1.17	5.18	0.31	4.27	**0.38**	**0.23**
16	2247.3	1773.5	1.17	4.05	0.31	2.28	**0.96**	**0.53**
32	563.2	437.1	1.17	**2.56**	0.31	**1.28**	2.89	1.71
64	141.9	109.9	1.17	**1.82**	0.31	**0.78**	11.26	5.91
128	36.75	28.25	1.17	**1.46**	0.31	**0.53**	42.41	21.75
256	9.67	7.46	1.17	**1.26**	0.31	**0.41**	176.22	86.53
512	2.58	1.99	1.17	**1.18**	0.31	**0.36**	753.81	348.8
1024	**0.75**	0.58	0.87	0.88	0.31	**0.32**	3116.7	1505.7
2048	**0.28**	**0.22**	0.32	0.33	0.28	0.29	21876.1	6027.8

Figure 4.4: Preprocessing overhead of communication set generation algorithms

'Method B naive' stands for the block-based communication generation method that computes all the intersection of all index sections of the work set with all index sections of the local index set. In the worst case an upper bound for the number of index sections of the local index set and the work index set is given by $\left\lceil \frac{N}{P.M} \right\rceil$. As a consequence, for a constant array size N the number of index sections doubles if the block size M is halved. Therefore, in this case the number of section intersections and thus the running time approximately quadruples. The numbers in Figure 4.4 clearly show that the overhead for this method is unacceptable high for small block sizes and large array sizes.

'Method B opt.' stands for the optimized block-based method. This algorithm computes the send index set in two phases. In the first phase the intersection of all sections of the relevant part of the local index set of B with all sections of the relevant part of the work index set of A is computed, yielding the minimal part of the send index set. The time required for these intersections is shown in Figure 4.4 in the column 'phase 1'. This time is constant for block-sizes from $M = 8$ to $M = 512$. In the case where $M = 4$ the number of index sections needed for the representation of the minimal part of the work set is smaller since some index sections are empty. Consequently, less section intersections have to be performed. In the case where $M \geq 1024$ the total number of index sections of the relevant part of the work set equals the total number of index sections of the whole work index set and thus the send index set is completely determined after phase 1. In the second phase the send index set is constructed based on the minimal part of the send index set according to Algorithm 4.2.7. The numbers in the column 'total' report the total time required for computing the send index set. Note that the complexity of the first phase of this algorithm is independent of the array size N.

'Method C' constructs the send index set according to the cyclic-based method of communication set generation as described in Section 4.2.4.1. The complexity of this algorithm solely depends on the number of indices contained in the minimal part of the local index set of B (i.e Σ^B) and the number of indices contained in the minimal part of the work index set of A (i.e $\mathcal{W}^A$). Note that both Σ^B and $\mathcal{W}^A$ are bound by the block-size M but are independent of the array size N. As the numbers show, the running time approximately quadruples if the block size M is doubled. As a consequence, this method introduces an unacceptable overhead for large block sizes (i.e. $M > 64$).

Our implementation currently does not yet handle all possible optimizations. The algorithms can be further optimized by using shift operations instead of divisions or multiplications if the number of processors or the block size are powers of 2.

4.2.4.4 Execution Schemes for Parallel Array Assignments

In the following we show how an array assignment statement of the above form can be transformed into SPMD pseudo-code. Furthermore, we discuss alternative approaches for cases where the *lhs* array section is replicated. It can be seen that in this case relaxing the owner computes paradigm can significantly reduce the communication overhead. Note however that these schemes are simplified since we do not consider here index conversion issues and buffering of messages. Code generation for array assignment statements will be discussed in more detail in Section 6.2.2.

Example 4.2.7: SPMD pseudo-code for array assignments

```
Code on processor p ...

Work^A(p) = Local^A(p) ∩ [l : u : s]
for each q ∈ P, q ≠ p do
  if Local^A(q) ≠ φ then
    Work^A(q) = Local^A(q) ∩ [l : u : s]
    if Work^A(q) ≠ φ then
      Send^B(p,q) = f(Work^A(q)) ∩ Local^B(p)
      if Send^B(p,q) ≠ φ then
         send Send^B(p,q) to q
      endif
    endif
  endif
endfor
if Work^A(p) ≠ φ then
  if Local^B(p) ≠ φ then
    Local_Work(p) = Work^A(p) ∩ f^{-1}(Local^B(p))
    execute local references
  end if
  for each q ∈ P, q ≠ p do
    Recv^B(p,q) = Send^B(q,p)
    if Recv^B(p,q) ≠ φ then
      receive Recv^B(p,q) from q
      execute non-local references
    endif
  endfor
endif
```

■

The code in Example 4.2.7 is somehow a worst case scenario, because each processor p has to determine the work set for all other processors. As a consequence, each processor must p must construct the local index sets of all other processors. Of course this general scheme has to be optimized in order to get reasonable efficiency. For many important special cases it is possible to determine at compile time that each processor only has to communicate with a small number of neighboring processors. For example, Koelbel [79] has worked out a number of special cases where the communication partners for a particular processor can be determined analytically. However, in this work neither block-cyclic distributions nor alignment is considered.

Utilizing Replication

If the left hand side variable of an array assignment statement is replicated, this can be utilized to simplify the above execution scheme. There are two approaches to do so. The first approach is strictly based on the owner computes paradigm. Since the left hand side array is replicated, the work set on each processor is given by the left hand side reference space and therefore need not be determined explicitly.

Another alternative in case of replication is to determine a unique master processor that is responsible for the evaluation of the whole assignment. After having finished the assignment the master processor has to broadcast the result to all other processors. Usually this approach will be more efficient since the number of messages will be smaller than in the first approach. In Example 4.2.8 these two schemes are compared.

Algorithm 4.2.8: Replication - Master Computes

```
if p ≠ master then
   Send^B(p, master) = f([l : u : s]) ∩ Local^B(p)
   if Send^B(p, master) ≠ φ then
      send Send^B(p, master) to master
   endif
endif
if p = master then
   Local_Work(p) = [l : u : s] ∩ f^{-1}(Local^B(p))
   perform local assignments
   for each q ∈ P, q ≠ master do
      Recv^B(master, q) = Send^B(q, master)
      if Recv^B(p, q) ≠ φ then
         receive Recv^B(p, q) from q
         perform non-local assignments
      endif
   endfor
   broadcast A
endif
```

Algorithm 4.2.9: Replication - Owner Computes

```
for each q ∈ P, q ≠ p do
   Send^B(p, q) = f([l : u : s]) ∩ Local^B(p)
   if Send^B(p, q) ≠ φ then
      send Send^B(p, q) to q
   endif
endfor
Local_Work(p) = [l : u : s] ∩ f^{-1}(Local^B(p))
perform local assignments
for each q ∈ P, q ≠ p do
   Recv^B(p, q) = Send^B(q, p)
   if Recv^B(p, q) ≠ φ then
      receive Recv^B(p, q) from q
      perform non-local assignments
   endif
endfor
```

Example 4.2.8: Replication: Master Computes vs. Owner Computes

```
PROCESSOR R(P)
REAL, DISTRIBUTED(BLOCK) :: B(N*P)
REAL                     :: A(N*P)

A = B
```

In case of the owner computes approach, the total number of messages that are generated during execution of the assignment is $(P-1)^2$*, whereas execution based on the master computes approach results in* $P-1$ *receive operations and one broadcast operation for the master processor and in one send and one receive operation on all other processors, which gives a total of* P *messages.* ∎

In the basic parallelization phase the communication that is necessary for an array assignment statement is represented by high level communication descriptors (see Section 4.4.2.1).

4.2.4.5 Element Based Communication

As already mentioned, communication sets that are generated because of assignments to array elements appearing in DO loops or FORALL statements may have to be determined on a single element basis. This is also the case for irregular array sections (e.g. vector subscripts). However, if the restrictions listed in Section 4.2.3.5 are fulfilled, communication sets for loops and FORALL statements can be handled in the same way as for array assignments.

4.2.4.6 Overlap Descriptions for Array Assignment Statements

If all arrays involved in an array assignment statement have a block or general block distribution, the overlap area concept for management of non-local data can be used. The overlap concept is applicable if it is possible to determine at compile time (eventually parameterized) the communication pattern generated by particular assignment statements. A major advantage of the overlap concept is that memory for non-local data can be allocated statically and computation of communication sets at run-time can be avoided. In order to determine overlap descriptions for array assignments, the communication index sets as defined in the previous sections can be utilized.

Under certain conditions the overlap concept can also be applied for arrays that have a block-cyclic distribution in order to optimize shift or stencil computations. Methods for the computation of overlap descriptions for block and block-cyclic distributions are discussed in detail in Section 5.4.1.

4.2.5 Handling Multidimensional Arrays

So far we have considered only one-dimensional arrays for the computation of the various sets. Local index sets and work index sets can be easily constructed by means of the Cartesian product of the corresponding sets in each dimension.

Let in the following $A(N_1, ..., N_n)$ and $B(N'_1, ..., N'_n)$ denote n-dimensional arrays that are distributed to a m-dimensional processor array R. Furthermore, let particular processors $p,\ q \in R$, be designated by the m-tuples $p = (p_1, \ldots, p_m)$ and $q = (q_1, \ldots, q_m)$, respectively.

The set of local indices $Local^A(p)$ of a multidimensional array A is given by:

$$Local^A(p) \quad = \quad Local^{A_1} \times \ldots \times Local^{A_n} \tag{4.114}$$

where for $1 < i < n$

$$Local^{A_i} \quad = \quad \begin{cases} Local^{A_i}(p_j) & \textit{if i-th dimension of A is distributed} \\ & \textit{to processor dimension j} \\ & \\ [1 : N_i] & \textit{otherwise} \end{cases}$$

The work set $Work^A(p)$ according to a multidimensional array section $A(l_1 : u_1 : s_1, \ldots, l_n : u_n : s_n)$ is given by:

$$Work^A(p) \;=\; Work^{A_1} \times \ldots \times Work^{A_n} \tag{4.115}$$

where for $1 < i < n$

$$Work^{A_i} \;=\; \begin{cases} Work^{A_i}(p_j) & \textit{if i-th dimension of A is distributed} \\ & \textit{to processor dimension j} \\ & \\ [l_i : u_i : s_i] & \textit{otherwise} \end{cases}$$

Similarly, the send index set for an array assignment statement of the form

$$A(l_1 : u_1 : s_1, \ldots, l_n : u_n : s_n) = B(\tilde{l_1} : \tilde{u_1} : \tilde{s_1}, \ldots, \tilde{l_n} : \tilde{u_n} : \tilde{s_n})$$

is given by:

$$send^B(p, q) \;=\; (f_1(Work^{A_1}) \cap Local^{B_1}) \times \ldots \times (f_n(Work^{A_n}) \cap Local^{B_n}) \tag{4.116}$$

where for $1 < i < n$, $1 < j < m$, $p = (p_1, \ldots, p_m)$, and $q = (q_1, \ldots, q_m)$,

$$Work^{A_i} \;=\; \begin{cases} Work^{A_i}(q_j) & \textit{if i-th dimension of A is distributed} \\ & \textit{to processor dimension j} \\ & \\ l_i : u_i : s_i & \textit{otherwise} \end{cases}$$

$$Local^{B_i} \;=\; \begin{cases} Local^{B_i}(p_j) & \textit{if i-th dimension of B is distributed} \\ & \textit{to processor dimension j} \\ & \\ [1 : N_i'] & \textit{otherwise} \end{cases}$$

with $f_i(j) = \frac{(j-l_i)}{s_i}.\tilde{s_i} + \tilde{l_i}$.

4.2.6 Runtime Representation of Distributions

In order to generate correct SPMD code, it is necessary to know for each reference to a distributed object the distribution that may reach this reference. However, since the distribution of an object may be changed dynamically, either through the execution of a distribute statement or as the result of a procedure call, multiple distinct distributions may reach a certain reference. Furthermore, dummy arguments may inherit the distribution of the actual arguments. As a consequence, dummy arrays may be associated with distinct distributions in different incarnations of a procedure.

In the case where different distributions reach a procedure, node splitting techniques could be applied. However, this approach is only feasible if the compiler can determine through interprocedural distribution analysis [49] that only a small number of different distributions may reach a procedure. This is also true for analysis methods like reaching distribution analysis and live distribution analysis [61].

Generally, the precise distribution of all objects can not be determined at compile time by any amount of analysis. As a consequence, it is necessary to have a runtime representation of distributions, called **layout descriptors**. Moreover, the compiler must parameterize the resulting target code. For example if different distributions may reach a procedure the resulting SPMD program has to be parameterized, by passing the actual arguments together with their layout descriptors. This however may have a critical impact on the efficiency of the resulting program.

A layout descriptor for a distributed array A must contain information about

- the global index domain, I^A
- the distribution function for each dimension, including the parameters of the distribution function
- the processor array(section) to which the array is distributed
- the alignment source array (if any)
- the alignment function (if any), of each dimension
- the local index set
- the active array section

Furthermore, a layout descriptor contains information that can be utilized to lower the runtime overhead of index conversion and communication generation. The information concerning the active array section is needed for the mechanisms of passing array sections to assumed shape dummy arrays, as explained later.

Example 4.2.9 shows the Fortran 90 type definition of a layout descriptor. In order to allow sharing of components, such as the information about the processor array or the alignment source array, it is defined in a modular way, consisting of pointers to sub-descriptors.

Usually a layout descriptor is initialized whenever an array is associated with a new distribution. In case of statically distributed arrays, the layout descriptor is initialized at program start. If specific information is available, parts of the descriptor may be initialized at compile time. In the following a layout descriptor of an array A will be designated by Δ^A.

Example 4.2.9: Layout Descriptors

```
TYPE PROC_DSC                                   ! processor array descriptor
     INTEGER                :: RANK
     INTEGER, POINTER       :: SHAPE(:)
     LOGICAL                :: SUBSET         ! to processor subset ?
     INTEGER, POINTER       :: DIM_PERM(:)    ! dimension permutation
     TYPE(TRIPLET), POINTER :: DIM_SEC(:)     ! dim. processor section
END TYPE PROC_DSC

TYPE DIM_DIST_DSC                               ! distribution descriptor
     INTEGER, POINTER :: HOW                    ! 0 ... replicated
                                                ! 1 ... BLOCK
                                                !   ...
                                                ! 6 ... CYCLIC(M)
     INTEGER, POINTER :: M(:)                   ! blocksize
     INTEGER, POINTER :: MAP(:)                 ! ptr to linearized Map array
END TYPE DIST_DSC

TYPE LID_DESC                                   ! dim. local index set descr.
     TYPE(TRIPLET_SET)      :: LOCAL          ! min. part of local index set
     LOGICAL                :: EMPTY          ! is anything there ?
     INTEGER                :: SIZE           ! total size of local index set
     ...
END TYPE LID_DESC

TYPE DIM_ALIGN_DESC
     INTEGER, POINTER         :: WITH_DIM     ! corr. target dimension
     INTEGER, POINTER         :: K            ! alignment stride
     INTEGER, POINTER         :: D            ! alignment offset
     INTEGER, LOGICAL         :: COLLAPSED    ! is dimension collapsed ?
END TYPE DIM_ALIGN_DESC

TYPE LY_DSC
     INTEGER                           :: RANK       ! rank of array
     INTEGER, POINTER                  :: GSHAPE(:)  ! global shape
     INTEGER, POINTER                  :: LSHAPE(:)  ! local  shape
     TYPE (DIM_DIST_DESC_PTR), POINTER :: DD_DSC(:)  ! dim. distribution descr.
     TYPE (LID_DESC_PTR), POINTER      :: LID_DSC(:) ! local index set descr.
     TYPE (PROC_DSC), POINTER          :: PROC       ! processor array descriptor
     LOGICAL                           :: ALIGNED    ! is array aligned ?
     TYPE (ALIGN_DSC_PTR), POINTER     :: DA_DSC(:)  ! dim. alignment descriptor
     TYPE (LY_DSC), POINTER            :: SOURCE     ! source array distr. descr
     TYPE (TRIPLET_LIST), POINTER      :: SECLIST(:) ! active section info
END TYPE LY_DSC
```

■

4.3 Overview of the Compilation Process

In this section we given an overview of the process of restructuring a Vienna Fortran 90 program into a data-parallel program which can be executed on a distributed memory multiprocessor. In particular we describe the main features of automatic parallelization by specifying a source-to-source translation from VF90 programs to optimized Fortran 90 message passing programs. Conceptually this process can be described as a sequence of four distinct phases, each of which specifies a translation between two source language levels. This is illustrated in Figure 4.5.

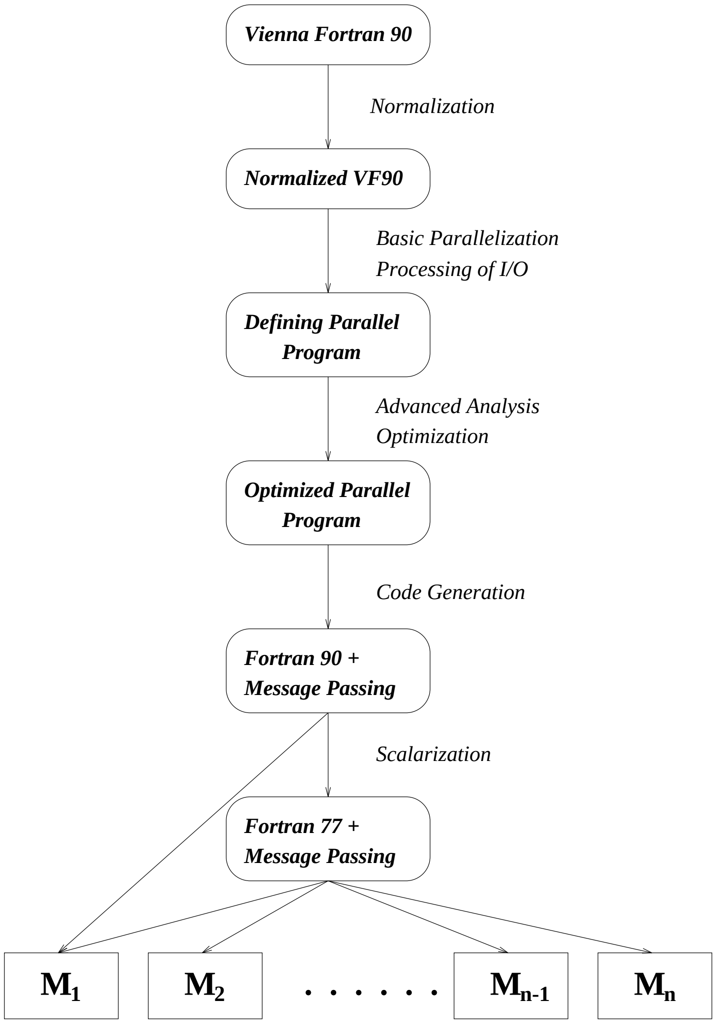


Figure 4.5: Compilation phases

Phase 1: VF90 Source Program → Normalized VF90 Program

In the first phase the source program is processed by the front end and transformed into an internal representation (i.e abstract syntax tree). Furthermore, in this phase initial program analysis and code normalization takes place. Normalizations have the purpose of generating a standard form of the program in order to simplify subsequent compiler operations. Normalization transformations typically include loop normalization, if conversion and subscript standardization. Array declarations, for example, are transformed in such a way that the lower bound in each dimensions is 1, and all corresponding references are transformed accordingly. Do loops are normalized such that the step size becomes 1. Subscript expressions are standardized in order to simplify data dependence analysis. Furthermore, standard transformations like constant propagation and dead code elimination are applied. For details about these transformations see for example [138]. A summary of program normalization is given in [12].

Initial program analysis includes syntactic and semantic analysis control flow analysis, data flow analysis, and data dependence analysis. Appropriate data structures which capture the analysis information are constructed. This includes the flow graph, the data dependence graph, and the call graph. The call graph is annotated by information concerning the bindings of actual and formal procedure parameters, which is needed in later phases.

Phase 2: Normalized VF90 Program → Defining Parallel Program

In the second phase, the **basic parallelization** phase, the normalized VF90 program is transformed into an explicitly parallel program, which is referred to as defining parallel program. The defining parallel program is based on the SPMD execution model which implies that each processor executes the same program. Therefore, the program has to be parameterized by the processor number. The defining parallel program is generated in three major steps. First the distribution and alignment specifications of the input program are processed in order to determine the ownership of all data objects.

In the second step, potential accesses to non-local data are satisfied by inserting *high level communication descriptors* prior to each assignment statement that contains references to distributed data objects. Communication descriptors are an abstraction of the actual communication that is necessary and are expanded into calls to appropriate message passing primitives in the code generation phase. The semantics of communication descriptors ensures that non-local data items are copied into private variables of the processor (i.e. temporary buffers).

In the third step work distribution is realized by associating each assignment statement with a mask (i.e. boolean guard) in order to enforce the owner computes paradigm. The mask ensures that each processor only updates those parts of an object that are owned by it, and that accesses to non-local data items are performed by accessing the corresponding private variables. If non-local data has to be accessed via temporary buffers, then the mask will be implemented in such a way that accesses to local data elements are split from accesses to non-local data.

Procedure calls are processed as follows. Whenever it is necessary to redistribute an actual argument in order to match the distribution of the formal argument, an assignment to a temporary array with the desired distribution is inserted before the call and the corresponding actual argument is replaced by the temporary array. In addition, an assignment of the temporary to the actual argument may have to be inserted after the procedure call. The newly generated assignment statements are processed in the same way as described above.

The program obtained after the basic parallelization phase specifies exactly the work distribution and communication required by the input program. However, it's performance is likely to be very poor, for several reasons. One reason is the run-time overhead that is required for evaluating the statement masks and the communication descriptors. Furthermore, there may be a large memory overhead since temporaries which have been introduced for non-local accesses are not reused. The aim of the subsequent program transformations therefore is to improve the parallel program wherever possible.

Phase 3: Defining Parallel Program → Optimized Parallel Program

The third phase, the optimization phase, aims to improve the efficiency of the parallel program by reducing the communication overhead and storage requirements and increasing the amount of useful work that may be performed in parallel by optimizing the work distribution.

Communication optimization is achieved by eliminating unnecessary or redundant communication descriptors, moving communication descriptors out of loops, combining messages from different communication statements, overlapping communication and computation, and exploiting collective communication primitives wherever possible. In cases where enough compile time information is available, the amount of buffer space needed for non-local accesses can be determined statically and minimal storage for distributed arrays can be allocated. The goal of overlap analysis is to detect simple regular communication patterns in order to improve the management of non-local data and the organization of communication.

In many cases the overhead of work distribution for array assignment statements can be minimized by eliminating statement masks and adapting the reference space of the corresponding array sections accordingly. The iteration space of loops can be reduced by propagating the information of statement masks to the loop bounds and stripmining the loop across the processors.

Phase 4: Optimized Defining Parallel Program → Message Passing Target Program

In the final phase the optimized defining parallel program is transformed into a Fortran 90 message passing program. Declarations of distributed objects are transformed in such a way that each processor only allocates those elements that are owned by it. As a consequence, all references to distributed objects might have to be transformed such that global indices are translated into corresponding local indices. Communication descriptors are expanded into calls to appropriate routines contained in the Vienna Fortran run-time library.

4.4 Basic Parallelization

4.4.1 Introduction

In this section we introduce the *basic parallelization scheme* which is the first phase in the parallelization process. The basic parallelization transforms a normalized Vienna Fortran program into a SPMD program without performing any sophisticated program analysis. As a consequence, the resulting SPMD program may be extremely inefficient. Aggressive program optimization techniques have to be applied in later phases in order to improve the efficiency of the parallelized program.

The main steps of the basic parallelization are the following:

- Processing of distribution and alignment specifications
- Transformation of executable statements
  - Communication insertion
    Handling of accesses to non-local references using buffers which are updated via message passing. In the basic parallelization scheme communication is described by high level communication descriptors.
  - Masking
    Enforce the owner computes rule to achieve work distribution according to the specified data distribution. Assignment statements are masked (guarded) to ensure that all variable updates are exclusively in the local memory of the executing processor.
  - Splitting of references to local and non-local data
- Modification of declaration statements
  Storage allocation for local parts of distributed objects in order to reflect data distribution.

4.4.2 High Level Communication Descriptors

In order to allow manipulation of data movement at a rather early stage of parallelization, it is necessary to explicitly describe possible data movements by means of a high level communication descriptors. A high level communication descriptor contains all information that is needed to determine which elements have to be communicated and how the communication is organized, i.e. information about temporary storage. An important aspect of the high level communication descriptor is the abstraction from the details of the actual management of temporary storage and communication. Communication descriptors must be flexible enough to allow the following operations on them:

- recognizing and removing redundant communication descriptors
- movement of descriptors (i.e. extraction from loops, across procedure boundaries)
- fusion of communication descriptors
- allow for either optimization of memory usage or optimization of communication
- splitting into the sending and receiving components

- uniform treatment of references to scalars, array elements, and array sections

Optimization of communication will be performed entirely on the level of communication descriptors. In the code generation phase communication descriptors are transformed to message passing code.

Communication descriptors are generated either because of assignment statements or procedure calls. In the case of a procedure call, a communication descriptor is inserted immediately before the call, whenever it is necessary to redistribute the actual argument in order to match the distribution of the formal argument (which may be specified explicitly), and after a procedure call if the distribution of the actual argument has to be restored after return from the call.

The high level communication descriptor COMM has the following structure:

$$\mathbf{COMM}([\Delta^D, < \vec{dst} >,]\ \Delta^S, < \vec{src} >, \mathcal{M})$$

with

- Δ^D, layout descriptor of the destination (*lhs*) object
  in case of a scalar object this parameter is not present
- $\vec{dst}$, the access pattern of the destination object
  in case of a scalar object this parameter is not present
- Δ^S, layout descriptor of the source (*rhs*) object
  designates either a distributed scalar or an array section
- $\vec{src}$, the access pattern of the source (*rhs*) object
- $\mathcal{M}$, communication storage (buffer, overlap)

A communication descriptor with all five parameters present is called *array communication descriptor* describing communication based on array sections. A *scalar communication descriptor* has only three parameters and describes element based communication.

4.4.2.1 Semantics of High Level Communication descriptors

Array Communication Descriptor

An array communication descriptor describes the communication that is induced by an assignment between two conformable array sections. This is now described precisely.

Let $\vec{lhs}$, $\vec{rhs}$ denote section subscripts of arrays A and B, respectively. Let M denote a temporary buffer, and let P denote the set of available processors, with arbitrary processors p, $q \in P$. An array communication descriptor of the form

$$\mathrm{COMM}(\Delta^A, < \vec{lhs} >, \Delta^B, < \vec{rhs} >, \mathrm{M})$$

which for example is generated in case of an array assignment statement $A(\vec{lhs}) = B(\vec{rhs})$ has the following semantics when evaluated on a particular processor p.[10]

[10] COMM must be evaluatable on every $p \in P$.

```
!  send part
if Local^B(p) ∩ rhs ≠ φ ∧ p = Master(Local^B(p))  then
    for each q ∈ P, q ≠ p  do
        if (Send^B(p,q) ≠ φ)  then
            M = B(send^B(p,q))
            send M to q
        endif
    endfor
endif

!  receive part
if Work^A(p) ∩ lhs ≠ φ  then
    for each q ∈ P, q ≠ p ∧ q = Master(Local^B(q))  do
        if (Recv^B(p,q) ≠ φ)  then
            receive M from q
        endif
    endfor
endif
```

Figure 4.6: Semantics of communication descriptors

The communication primitive `send` is non-blocking and `receive` is blocking. Furthermore, we assume that different messages from a single sender to a single receiver arrive in the same order as they have been sent. The function *Master* returns the processor id of the master of the corresponding local index set. Therefore, in case of replication data is sent by only one uniquely defined processor.

Scalar Communication Descriptor

In case of an assignment to a scalar variable, the destination parameters of a communication descriptor are omitted[11]. In this case, the computation of the communication sets can be simplified and its semantics can be easily derived from above by utilizing the fact that the term $f(Local^A(q) \cap l\vec{hs})$ reduces to $r\vec{hs}$ and hence $Send^B(p,q)$ is given by $Local^B \cap r\vec{hs}$. Therefore, a scalar communication descriptor of the form

$$\mathrm{COMM}(\Delta^B, < r\vec{hs} >, \mathrm{M})$$

which for example is generated in case of a scalar assignment statement[12] $S = B(r\vec{hs})$ has the following semantics when evaluated on a particular processor p.

[11] This is based on the assumption that scalars are replicated on all processors.

[12] In this case, $r\vec{hs}$ designates an array element.

```
!  send part
if  (Local^B(p) ∩ r⃗hs ≠ φ ∧ p = Master(Local^B(p)))  then
   M = B(g2l(r⃗hs))
   for  each  q ∈ P, q ≠ p  do
      send  M  to  q
   endfor
endif

!  receive part
for  each  q ∈ P, q ≠ p ∧ q = Master(Local^B(q))  do
   if  (Local^B(p) ∩ r⃗hs ≠ φ)  receive  M  from  q
endfor
```

Figure 4.7: Scalar communication descriptors

4.4.3 Work Distribution

Work distribution is derived automatically by applying the *owner computes rule*. In the basic parallelization phase this is achieved by masking assignment statements so that only those processors that own the *rhs* reference execute the assignment. For this purpose the elemental logical function *OWNED* is used.

> Syntax: **OWNED** (*ref*, [p])
>
> *ref* may be scalar or array valued. The result is either scalar or array valued of type logical. If *ref* is scalar valued, the function returns the value *.TRUE.* if the array element denoted by *ref* is owned by processor p. The default for p is the id of the executing processor. If *ref* is array valued, the function result is a logical array with the same shape as *ref*.

The function *Owned* will be utilized in order to enforce the owner computes rule during the basic parallelization phase by guarding each assignment statement that references distributed objects by a conditional statement. This is discussed in more detail later.

In order to guarantee the correctness of the basic parallelization transformations, it is necessary that the program is normalized as follows:

- Transformation of Function Calls
  Since a function may have side effects, it is necessary to enforce that all processors execute the function call in order to preserve the semantics of the program. In the basic parallelization phase expressions containing references to scalar functions are transformed by substituting the function reference by a new temporary scalar variable and generating an assignment statement which assigns the function result to the temporary. A reference to an array valued function will be transformed into a subroutine call, as it can be seen in Example 4.4.1.

- Transformation of Indirect Array References
  Index expressions in references to distributed arrays must exclusively access replicated variables, i.e must never include references to distributed arrays. This restriction can be enforced by a simple transformation that extracts references to distributed arrays from such index expressions. Consider a reference of the form *A(X(i))*, where A and X are distributed arrays. This is transformed by inserting an assignment of $X(i)$ to a new scalar variable prior to the original statement and replacing the reference in the index expressions accordingly. If X is used as a vector subscript in the reference *A(X)*, then X is assigned to a replicated temporary array prior to the original statement and the reference is replaced in the index expression by this temporary array.

Example 4.4.1: Transformation of expressions referencing array valued functions

original code

```
REAL A(N)
DISTRIBUTED(δ^B) ::  B
...
INTERFACE
   FUNCTION F(X)
   REAL, ::  X(:)
   REAL, DISTRIBUTED(δ^F) ::  F(SIZE(X))
   ....
END INTERFACE


... + F(A) + ...
```

transformed code

```
REAL A(N)
REAL, ALLOCATABLE ::  tmp_F(:)
...
INTERFACE
   SUBROUTINE F(tmp_F,X)
   REAL X(:), tmp_F(:)
   DISTRIBUTED(δ^F) ::  tmp_F
   ....
END INTERFACE
...
ALLOCATE(tmp_F(SIZE(A)))
CALL F(tmp_F,A)
... + tmp_F + ...
```

Note that the Fortran 90 standard requires that a function must have an explicit interface if it has an array-valued result. ■

4.4.4 Transformation of Executable Statements

In this section we show how executable statements are transformed during the basic parallelization phase. Each statement is masked in order to ensure that a processor only computes those elements which are in its local memory. Potential movement of non-local data is ensured by inserting high level communication descriptors for each reference to a distributed array in the *rhs*. So the transformation of executable statements compromises masking and communication insertion. Note that CALL statements play a special role because a CALL is always performed by all processors, and therefore is not masked during basic parallelization.

4.4.4.1 Scalar Assignments

In the following we distinguish between *replicated scalars* which are ordinary scalars in the sense of Fortran 90 and *distributed scalars* which correspond to elements of distributed arrays. An assignment statement is transformed according to the following rules:

- If a distributed scalar appears on the right hande side of a scalar assignment statement, a communication descriptor is inserted in order to enforce that its value is communicated to those processors which do not own this distributed scalar. Obviously, this will usually result in a broadcast.
- If a distributed scalar appears on the left hand side of a scalar assignment, the statement is masked in order to enforce the *owner computes* rule.
- A scalar assignment involving only replicated scalars is left unchanged.

Communication descriptors which are generated during basic parallelization of scalar assignments usually have three parameters only. All the possible cases of scalar assignments that may arise are shown in Example 4.4.2.

Let i, j denote arbitrary Fortran 90 subscripts (i.e. scalar integer expressions). Statement s1 needs no transformation because no distributed objects are involved. In statement s2 an element of a distributed array is assigned to a scalar. Therefore, a scalar communication descriptor is generated to ensure that the value of the lhs array element is made available on each processor. In the final code this usually will result in a broadcast operation. Statement s3 is masked by means of an *IF* statement in order to ensure that only those processors which are owners of A(i) execute the assignment. Transformation of s4 introduces a communication descriptor as well as a scalar mask. How the mask is actually implemented is dependent on the method used for handling accesses to non-local data.

Example 4.4.2: Basic parallelization of scalar assignments

original code

```
      REAL S, T, A(N), B(M)
      DISTRIBUTED(somehow) ::  A, B

   s1:T = S

   s2:S = A(j)

   s3:A(i) = T

   s4:A(i) = B(j)
```

transformed code

```
      !  not transformed
   s1:T=S

      !  insert COMM
   s2:COMM (Δ^A, <j>, tmp)
      S = tmp

      !  insert mask
   s3:IF  (Owned(A(i))) A(i) = T

      !  insert COMM and mask
   s4:COMM (Δ^A, <i>, Δ^B, <j>, tmp)
      Mask  → A(i) = B(j)
```

■

The default mechanism for management of non-local data is based on temporary buffers. As a consequence, it is necessary to split local and non-local accesses. Therefore, the mask of a scalar assignment statement usually will be transformed as shown in Example 4.4.3, where *tmp* is the temporary variable which is updated by the corresponding communication descriptor.

Example 4.4.3: Semantics of scalar masks

The masked scalar assignment

Mask $\rightarrow$ `A(i) = B(j)`

is transformed to

```
IF (Owned(A(i))) THEN
   IF (Owned(B(j))THEN
      A(i) = B(j)     ! local reference
   ELSE
      A(i) = tmp      ! non-local reference
   END IF
END IF
```

■

In special cases, access to non-local data can be handled by means of the overlap concept. If an assignment statement is parallelized based on the overlap concept, splitting of local and non-local references can be avoided. The overlap concept and its integration with the method of temporary buffers is discussed in Section 5.4 and Section 6.1.6.

4.4.4.2 Array Assignments

Array assignment statements are transformed during the basic parallelization stage conceptually in the same way as scalar assignments. However, communication descriptors and masks are based on array sections rather than on array elements. Communication descriptors that are generated in the basic parallelization phase of array assignments contain all the information that is needed for the calculation of work sets and communication sets using the formulas derived in Sections 4.2.3 and 4.2.4.

For an array assignment statement $A(\vec{lhs}) = B(\vec{rhs})$ a communication descriptor contains the following information:

- Δ^A, layout descriptor of the *lhs* object
- $\vec{lhs}$, the access pattern of the *lhs* section
- Δ^B, layout descriptor of the *rhs* object
- $\vec{rhs}$, the access pattern of the *rhs* section
- name of the buffer

During the basic parallelization it is assumed that non-local elements are managed via temporary buffers. It is the task of the later optimization phases to replace buffers by more efficient schemes, as overlaps, wherever possible.

In the following we distinguish three different cases of array assignments, depending on the *rhs* object which can be a replicated scalar, a distributed scalar, or an array section. All possible cases that may arise during basic parallelization of array assignment statements are shown in Example 4.4.4.

Example 4.4.4: Basic parallelization of array assignment statements

original code

```
REAL A(N), B(M), S
DISTRIBUTED(somehow) ::  A,B

s1:A(lhs) = S

s2:A(lhs) = B(i)

s3:A(lhs) = B(rhs)
```

transformed code

s1:WHERE (Owned(A($l\vec{hs}$))) A($l\vec{hs}$) = S

s2: **COMM** (Δ^A, $<l\vec{hs}>$, Δ^B, $<i>$, tmp)
Mask $\rightarrow$ A($l\vec{hs}$) = B(i)

s3: **COMM** (Δ^A, $<l\vec{hs}>$, Δ^B, $<r\vec{hs}>$, tmp_B)
Mask $\rightarrow$ A($l\vec{hs}$) = B($r\vec{hs}$)

■

Usually the mask of an array assignment statement can be transformed by introducing two *WHERE* statements as shown in Example 4.4.5. One *WHERE* statement implements local accesses and the other realizes the access to non-local data stored in a temporary buffer.

Example 4.4.5: Semantics of array masks

The masked array assignment

Mask $\rightarrow$ A($l\vec{hs}$) = B($r\vec{hs}$)

is transformed to

```
!  local references
WHERE (Owned(A(lhs)) .AND. Owned(B(rhs))) A(lhs) = B(rhs)
!  non-local references
WHERE (Owned(A(lhs)) .AND..NOT. Owned(B(rhs))) A(lhs) = tmp_B
```

■

If on the right hand side more than one distributed array appears, a communication descriptor has to be generated for each of those arrays. However, in the case of multiple array references appearing on the right hand side, splitting of the array assignment statement into local and non-local parts is more complex. Assume that we have the following array assignment statement

A($l\vec{hs}$) = B_1($r\vec{hs}_1$) + ... + B_n($r\vec{hs}_n$)

The local work set for this assignment statement on a particular processor $p \in P$ is given by

$$Local\_Work(p) \quad = \quad Work^A_{\vec{I}}(p) \cap f_1^{-1}(Local^{B_1}(p)) \cap \ldots \cap f_n^{-1}(Local^{B_n}(p))$$

where $f_i^{-1}, 1 \leq i \leq n$, is the linear function that maps the elements of $A(\vec{lhs})$ to the corresponding elements of $B_i(\vec{rhs}_i)$ (see Section 4.2.4).

All other instances of the work set correspond to non-local references. As can be seen, the non-local references on a particular processor p now access not only non-local elements of the arrays B_i, which are received from other processors and stored in temporary buffers, but also elements that are in the local memory of p. As a consequence, the non-local references have to be split into a series of WHERE statements as shown below.

Example 4.4.6: Basic parallelization of array assignments with multiple *rhs* arrays

```
! comm descriptor for each distributed rhs object
COMM (Δ^A, <lhs>, Δ^B1, <rhs_1>, tmp_B1)
        ...
COMM (Δ^A, <lhs>, Δ^Bn, <rhs_n>, tmp_Bn)

! local references
WHERE (Owned(A(lhs)) .AND. Owned(B1(rhs_1)) ...  .AND. Owned(Bn(rhs_n))) &
   A(lhs) = B1(rhs_1) + ... + Bn(rhs_n)

! non-local references
WHERE (Owned(A(lhs)) .AND. Owned(B1(rhs_1)) ...  .AND..NOT. Owned(B(rhs_n))) &
   A(lhs) = B1(rhs_1) + ... + tmp_Bn
        ...
WHERE (Owned(A(lhs)) .AND..NOT. Owned(B1(rhs_1)) ...  .AND. Owned(B(rhs_n))) &
   A(lhs) = tmp_B1 + ... + Bn(rhs_n)

WHERE (Owned(A(lhs)) .AND..NOT. Owned(B1(rhs_1)) ...  .AND..NOT. Owned(B(rhs_n))) &
   A(lhs) = tmp_B1+ ... + tmp_Bn
```

■

This approach, however, is only feasible if the number of arrays with different distributions appearing in the *rhs* expression is small. If this is not the case, there are two alternative approaches. The first one is to split the expression into smaller ones by introducing temporary variables, which are aligned with the left hand side array. This would allow to maximize the local work set and to simplify code generation, however, with the drawback of additional memory requirement. The other possibility is to give up splitting of local and non-local references. This however requires that the elements local to a processor are copied into a temporary buffer as well.

4.4.4.3 Conditional Statements

Basic parallelization of *IF* statements is performed in two steps:

- processing of the condition
- processing of the action statement

Communication descriptors may be generated for the evaluation of the condition as well as for the action statement.

If the condition of the *IF* statement, which is a scalar logical expression, contains a reference to a distributed array, the *IF* statement is transformed as follows. A new temporary scalar logical variable, say *t_cond*, is introduced in order to hold the result of the condition. Before the *IF* statement, an assignment of the scalar logical expression to *t_cond* is inserted and the condition is replaced by *t_cond*. Since *t_cond* is replicated, all processors will take the same control flow. The generated assignment statement is then treated analogously to a scalar assignment, which results in the generation of a communication descriptor. If the condition of the *IF* statement does not contain a reference to a distributed array, no such transformation takes place. In the second step the action statement is transformed by inserting communication descriptors and generating a mask. Communication descriptors (if any) are moved out from the *IF* statement. Furthermore, the mask of the action statement (if any) is moved to the *IF* statement in order to allow for later optimizations based on mask propagation.

Example 4.4.7: Basic parallelization of *IF* statements

```
IF (L(i1)) A(i2) = B(i3)
```

is transformed into

```
COMM(Δ^L, < i1 >, tmp_L)
t_cond = temp_L
COMM(Δ^B, < i3 >, temp_B)
IF(Owned(A(i2))) THEN          !  enforce owner computes rule
   IF (t_cond) A(i2) = B(i3)   !  local reference
ELSE
   IF (t_cond) A(i2) = temp_B !  non-local reference
ENDIF
```

■

IF constructs and *CASE* constructs are parallelized in a similar way as IF statements. If conditions of the *IF* construct contain references to distributed variables, new temporary scalar logical variables are introduced in order to hold the results of the conditions. Prior to the *IF* construct, assignments of scalar logical expressions to these temporaries are inserted and the conditions are replaced by temporaries. Since temporaries are replicated, all processors will take the same control flow. The generated assignment statements are then treated analogously to scalar assignments, which results in the generation of communication descriptors. In the second step blocks of statements put into *THEN* and *ELSE* clauses are transformed by inserting communication descriptors and generating masks.

When parallelizing *CASE* constructs types of the temporaries must equal the types of corresponding case-expressions and case-selectors [72].

4.4.4.4 Masked Array Assignment

The *WHERE* statement and the *WHERE* construct are used to control the execution of an array assignment statement with a logical mask expression. All arrays involved in a masked array assignment must conform in shape. We discuss here only the basic parallelization of *WHERE* statements. We assume that *WHERE* constructs are transformed into a series of *WHERE* statements during the program normalization phase.

A *WHERE* statement has to be transformed in the basic parallelization phase only if distributed arrays are involved. Basic parallelization of *WHERE* statements is based on the following general syntactical form.

WHERE (*logical-array-expr*) *array-assignment-stmt*

Conceptually, the execution of a masked array assignment can be split into two phases. Phase 1 performs mask evaluation and phase 2 executes the array assignment statement under control of the mask. Since distributed arrays can appear in the mask and/or in the array assignment statement, in both of these phases communication may be necessary. Obviously, parallelization of the array assignment statement is based on the distribution of the left hand side array. Goal of the basic parallelization of *WHERE* statements is to avoid communication for arrays involved in the mask evaluation at the time when the assignment statement is executed.

Therefore, it is necessary to enforce that the mask is aligned with the array on the left hand side of the array assignment statement. This is achieved by generating a temporary mask array, which is aligned with the array on the left hand side. Prior to the *WHERE* statement an assignment to the temporary mask array is inserted. This is shown in the following example.

Example 4.4.8: Basic parallelization of *WHERE* statements

original code

```
!  A, B, and MSK are distributed arrays
WHERE (MSK(msk⃗)...)  A(lhs⃗) = B(rhs⃗)
```

Step 1 - Transformation

```
LOGICAL, ALLOCATABLE, ALIGNED(tmp_Mask WITH A) ::  tmp_Mask(:)
....
ALLOCATE(tmp_Mask(SIZE(A)))
tmp_Mask(lhs⃗) = MSK(msk⃗) ...
WHERE (tmp_Mask(lhs⃗)) A(lhs⃗) = B(rhs⃗)
```

Step 2 - Basic parallelization of the transformed WHERE *statement*

```
....
!  evaluate distributed mask expression
COMM (\Delta^A, < \vec{lhs} >, \Delta^M, < \vec{msk} >, tmp_M)
WHERE (Owned(tmp_Mask(\vec{lhs})) .AND. Owned(M(\vec{msk}))) tmp_Mask(\vec{lhs}) = M(\vec{msk})
ALLOCATE (M(SIZE(tmp_Mask(\vec{lhs})))
M = Owned(tmp_Mask(\vec{lhs})) .AND..NOT. Owned(M(\vec{msk}))
WHERE ( M ) tmp_Mask(\vec{lhs}) = UNPACK( tmp_M, MASK = M, FIELD = .FALSE. )

!  execute transformed WHERE stmt
COMM (\Delta^A, < \vec{lhs} >, \Delta^B, < \vec{rhs} >, tmp_B)
WHERE (tmp_Mask(\vec{lhs}) .AND. Owned(A(\vec{lhs})) .AND. Owned(B(\vec{rhs}))) A(\vec{lhs}) = B(\vec{rhs})
M = tmp_Mask(\vec{lhs}) .AND. Owned(A(\vec{lhs})) .AND..NOT. Owned(B(\vec{rhs}))
WHERE ( M ) A(\vec{lhs}) = UNPACK( tmp_B, MASK = M, FIELD = 0 )
DEALLOCATE(tmp_mask, M)
```

■

Note that basic parallelization of the *WHERE* statement might introduce unnecessary communication since elements of B might be communicated although they are not needed because the corresponding elements of the mask are .FALSE..

If a non-elemental function reference occurs in the *rhs* of the masked array assignment and argument expressions of this function contain distributed objects, the function must be evaluated before the *WHERE* statement.

Example 4.4.9: *WHERE* statement containing non-elemental function reference

original code

```
!  A is distributed

WHERE (A > 0) A = A /SUM(LOG(A))
```

transformed code

```
ALLOCATE (T)
T = SUM(LOG(A))
WHERE (A > 0) A = A /T
DEALLOCATE (T)
```

■

4.4.4.5 CALL Statements and Procedure Interfaces

The *SPMD paradigm* is applied to procedures in an obvious way: a procedure call in the node program is executed by all processors allocated to the program. We assume that for every procedure the interface is explicit to the caller. A dummy argument may either have a distribution specified explicitly or it may inherit the distribution from the actual argument. In addition, for dummy arrays with inherited distribution, a distribution range may be specified. Different basic parallelization techniques are applied depending on whether the dummy arrays are of *assumed shape* or *explicit shape*.

4.4.4.5.1 Argument Passing via Assumed Shape Arrays

For argument passing via assumed shape arrays the Fortran 90 standard requires that actual and formal argument must have the same rank. Since the extents of assumed shape arrays must not be specified explicitly, there is always shape conformance between actual and formal argument.

Assumed Shape Dummy Arrays with Assumed Distribution

If a dummy argument array has an inherited distribution, then it can be bound to an actual argument of any distribution, so that in general it is not necessary to redistribute the actual argument. As a consequence, dummy arrays with inherited distribution can have a different distribution in each incarnation of the procedure. If it can be determined at compile time, either because a distribution range has been specified, or as the result of interprocedural distribution analysis, that only a small number of different distributions reach a procedure, then procedure cloning can be applied.

In the general case, however, the procedure has to be parameterized in order to handle different distributions. This is achieved by modifying the procedure interface such that each distributed array is passed together with its descriptor. A layout descriptor contains all information concerning the distribution of the actual array and about the transferred section. If the actual array is an array section rather than a "whole" array, information about the (extended) index domain of the passed array section is necessary in order to determine the distribution of the dummy argument. The argument transfer mechanism for assumed shape arrays with inherited distribution is always *call by reference*. However, if array sections are transferred some overhead may be introduced.

Basic parallelization of CALL statements for procedures with assumed shape dummy arrays with inherited distribution is performed in the following steps:

- Update Distribution Descriptor
  The section information in the layout descriptor of the actual argument is updated before and after the call by inserting or deleting the description of the transferred actual array section to the active section list.
- Modify Argument List
  Modify the CALL statement and the procedure header by adding the layout descriptor to the argument list.

During basic parallelization CALL statements and procedure interfaces are transformed as shown in Example 4.4.10. Let A be a distributed array and let $\vec{sec}$, $\vec{sec}_1$, $\vec{sec}_2$ denote section subscripts. The distribution function $\delta^Y(p)$ on a particular processor for array Y in subroutine $Sub2$ can be described in terms of the distribution of A as follows: Let t_1 denote the function that maps indices of the array section $A(\vec{sec}_1)$ to the index domain of the corresponding dummy array X and let t_2 denote the function that maps indices of the array section $X(\vec{sec}_2)$ to the index domain of dummy array Y. Then δ^Y is given by

$$\delta^Y(k) \quad = \quad \delta^X(t_1^{-1}(k)) = \delta^A(t_1^{-1}(t_2^{-1}(k)))$$

with $i \in \mathbf{I}^A$ and $k \in \mathbf{I}^Y$.

Example 4.4.10: Assumed shape inherited distribution dummy arrays

original code

```
REAL, DIMENSION(N_1, ...  ,N_n) :: A
DISTRIBUTED δ^A :: A
...
CALL Sub1(A(sec))
...
CONTAINS
   SUBROUTINE Sub1(X)
      REAL, DISTRIBUTED(*) :: X(:, ...  ,:)
      CALL Sub2(X(sec_1))
   END SUBROUTINE Sub1

   SUBROUTINE Sub2(Y)
      REAL, DISTRIBUTED(*) :: Y(:, ...  ,:)
      Y(sec_2) = ....
   END SUBROUTINE Sub2
END PROGRAM
```

transformed code

```
...
InsertSection(Δ^A, sec)       !  insert section information in descriptor
CALL Sub1(A(sec), Δ^A)
DeleteSection(Δ^A)
...
CONTAINS
   SUBROUTINE Sub1(X, Δ^X)
      ...
      InsertSection(Δ^X,sec_1) !  insert section information in descriptor
      CALL Sub2(X(sec_1), Δ^X)
      DeleteSection(Δ^X)
   END SUBROUTINE Sub1

   SUBROUTINE Sub2(Y, Δ^Y)
      ...
      COMM (Δ^Y, < sec_2 >, ...  )
      Mask  → Y(sec_2) = ....
   END SUBROUTINE Sub2
END PROGRAM
```

In Figure 4.8 the transfer of array sections is illustrated for an array $A(1:15)$ which is distributed by block to 3 processors. In the first call the section $A(1:15:2)$ is passed and in the second call the section $X(2:8:3)$ is passed. The inverse transfer functions therefore are given by $t_1^{-1}(i) = 2*i-1$ and $t_2^{-1}(i) = 3*i-1$, and for example $\delta^Y(2) = \delta^A(t_1^{-1}(t_2^{-1}(2))) = \delta^A(9)$. The function `InsertSection` *inserts information about the transferred array section in the list of active sections of the layout descriptor.*

■

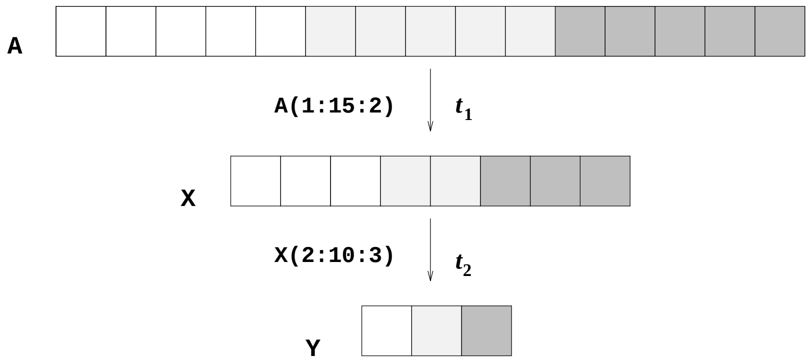


Figure 4.8: Assumed shape and assumed distribution argument transfer

Assumed Shape Dummy Arrays with Explicit Distribution

If the distribution of a dummy argument array is explicitly specified, then the actual argument has to be redistributed, if necessary, to match it. In this case, argument passing can no longer be call by reference. A *copy-in / copy-out* semantics must be adopted. In order to allow for later optimization based on communication descriptor movement, redistribution is performed by the caller before and after the call to the procedure.

A temporary array with the same size and shape as the actual array (section) is allocated with the desired distribution. The actual array (section) is assigned to that temporary, provided that neither the *NOTRANSFER* nor the *INTENT(OUT)* attribute has been specified for the dummy array. Since this assignment may result in communication, a high level communication descriptor is generated. In the *CALL* statement the original actual argument is replaced by the temporary. After the *CALL* statement the temporary array may have to be reassigned to the original array. This again may cause insertion of a communication descriptor provided the *INTENT(IN)* attribute has not been specified.

Basic parallelization of calls to procedures containing assumed shape arguments with explicit distribution comprise (see Example 4.4.11):

- Redistribution of Actual Argument Before Call
  This is achieved by allocating a temporary array of size and shape of the actual array (section) with distribution according to the formal array (section) and generating an assignment of the actual array (section) to the temporary.

- Modification of *CALL* Statement
  Pass temporary instead of original array (section).

- Redistribution of Actual Argument After Call
  If necessary, the temporary array has to be assigned to the original array section on exit from the call. The temporary array then can be deallocated.

Example 4.4.11: Assumed shape explicit distribution dummy arrays

original code

```
REAL, DISTRIBUTED(δ^A) ::  A(N_1, ...  ,N_n)
...
CALL Sub(A(sec⃗))
...
CONTAINS
   SUBROUTINE Sub(X)
      REAL, DISTRIBUTED(δ^X) ::  X(:, ...  ,:)
      ...
   END SUBROUTINE Sub
END PROGRAM
```

code after basic parallelization

```
REAL, DISTRIBUTED(δ^A) ::  A(N_1, ...  ,N_n)
REAL, ALLOCATABLE, DIMENSION(:, ...  ,:)  ::  T
...
!  allocate T with distribution δ^X
ALLOCATE(T(SIZE(A(sec⃗),1), ...  , SIZE(A(sec⃗),n)))

!  assign to temporary
COMM_1(Δ^T, <[:,...,:]>, Δ^A, <sec⃗>, tmp_A)
Mask_1 → T = A(sec⃗)

CALL Sub(T)

!  reassign from temporary (if needed)
COMM_2(Δ^A, <sec⃗>, Δ^T, <[:,...,:]>, tmp_T)
Mask_2 → A(sec⃗) = T
DEALLOCATE(T)
...
CONTAINS
   !  stays unchanged
   SUBROUTINE Sub(X)
      ...
   END SUBROUTINE Sub
END PROGRAM
```

■

4.4.4.5.2 Parameter Passing via Explicit Shape Arrays

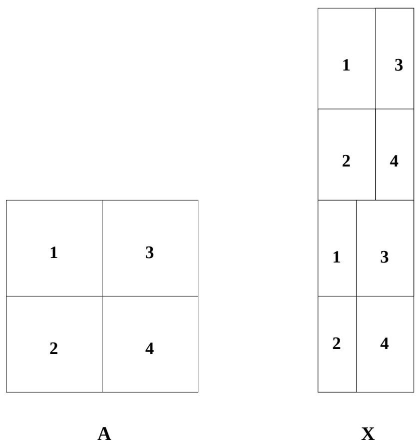
Argument passing using explicit shape dummy arrays is more complex than via assumed shape arrays because the restriction that the rank and shape of actual and formal array must be the same is absent. Therefore, we must be able to handle cases where a n-dimensional actual array is associated with a m-dimensional dummy array. Explicit shape dummies, as an old FORTRAN concept, imply contiguous storage and rely on storage association between actual and formal array. Because the actual argument can be an array section we are forced to adopt in the basic parallelization phase the *copy-in / copy-out* argument transfer mechanism. Generally, if an array section is passed to an explicit shape dummy the array section has to be copied into a contiguous temporary array. This temporary is then passed to the procedure and assigned back to the original array section on exit form the procedure if necessary. The actual argument must be redistributed, if necessary, to match the distribution of the dummy argument. Redistribution is achieved by allocating and assigning a temporary with the desired distribution. However, size and shape of this temporary are now determined by the dummy array rather than by the transferred section.

Explicit Shape Dummy Arrays with Assumed Distribution

If it can be determined at compile time that actual and formal argument always conform in shape, then the techniques as described in Section 4.4.4.5.1 are applied. If there is no shape conformance, the distribution of the dummy array might not be expressible via an intrinsic distribution function (except INDIRECT) or alignment, as shown in the following example.

Example 4.4.12: Reshaping of array arguments at procedure boundaries

```
PROCESSORS R(2,2)
REAL, DISTRIBUTED(BLOCK,BLOCK) ::  A(10,10)
   ...
   CALL Sub(A)
   ...
   SUBROUTINE Sub(X)
      INTEGER, DISTRIBUTED(*) ::  X(20,5)
      ...
   END SUBROUTINE Sub
```



In this example array A is associated with a dummy array X of different shape. As a consequence, the distribution of X can no longer be described as a composite distribution by means of intrinsic distribution functions. ∎

To handle such case where arrays get reshaped at procedure boundaries a default distribution for the dummy array is generated by the compiler.

Explicit Shape Dummy Arrays with Explicit Distribution

If the distribution of a dummy argument array is specified explicitly, then the actual argument has to be redistributed, if necessary, to match it. This enforces a *copy-in / copy-out* argument transfer.

Again redistribution is achieved by allocating a temporary with the corresponding distribution. However, size and shape of this temporary are now determined by the dummy array rather than by the transferred section.

Before each *CALL* statement to such a procedure an assignment of the original argument to the temporary array has to be generated. For those cases where the rank of the actual and formal argument differ, this assignment is realized using the Fortran 90 *RESHAPE* function. The *CALL* statement has to be adjusted accordingly, and after the *CALL* statement the temporary array has to be assigned back to the original argument, provided the *INTENT(IN)* attribute has not been specified.

Example 4.4.13: Explicit shape explicit distribution dummy arrays

original code

```
REAL, DISTRIBUTED( δ^A ) ::  A(N_1, ...  ,N_n)
...
CALL Sub(A(sec⃗), ...  )
...
CONTAINS
   SUBROUTINE Sub(X, ...  )
      REAL, DISTRIBUTED(δ^X) ::  X(M_1, ...  ,M_m)
         ...
   END SUBROUTINE Sub
```

transformed code

```
REAL, DISTRIBUTED(δ^A) ::  A(N_1, ...  ,N_n)
REAL, ALLOCATABLE, DIMENSION(:_1, ...  ,:_m) ::  T
...
!  allocate T with distribution δ^X, M'_i = M_i, i=1,...,m
ALLOCATE(T(M'_1, ...  ,M'_m))
T = RESHAPE(A, SHAPE(A(sec⃗)))
CALL Sub(T, ...  )
A(sec⃗) = RESHAPE(T, SHAPE(T))
DEALLOCATE(T)
...
CONTAINS
   !  stays unchanged
   SUBROUTINE Sub(X, ...  )
      ...
   END SUBROUTINE Sub
END PROGRAM
```

■

Basic parallelization of calls to procedures containing explicit shape arguments with explicit distribution as illustrated in Example 4.4.13 comprises:

- Redistribution of Actual Argument Before Call
  This is achieved by allocating a temporary array with size, shape and distribution equal to the dummy array and generating an assignment of the actual array (section) to the temporary.
- Modification of *CALL* Statement
  Pass temporary instead of original array (section).
- Redistribution of Actual Argument After Call
  If necessary, the temporary array has to be assigned to the original array section on return from the call. The temporary array then can be deallocated.

4.4.4.5.3 Scalar Dummy Arguments

So far we have considered only procedure arguments that are arrays. If an actual argument that is a distributed scalar (i.e. element of a distributed array) is associated with a scalar dummy argument, the actual argument has to be broadcast. This is achieved by introducing a temporary scalar variable, generating an assignment of the distributed scalar to the temporary, and passing the temporary instead of the distributed scalar to the procedure. On exit from the procedure the temporary variable must be assigned back to the original distributed scalar, if the dummy has not got the *INTENT(IN)* attribute. The code of the procedure needs no modification.

Example 4.4.14: Procedures with scalar dummy

original code

```
DISTRIBUTED(δ^A) ::  A
...
CALL Sub(A(i))
...
CONTAINS
   SUBROUTINE Sub(S)
   REAL ::  S
      ...
   END SUBROUTINE Sub
END PROGRAM
```

code after basic parallelization

```
DISTRIBUTED(δ^A) ::  A
...
COMM(Δ^A,< i >,tmp)
CALL Sub(tmp)
IF (Owned(A(i))) A(i) = tmp
...
CONTAINS
   !  stays unchanged
   SUBROUTINE Sub(S)
      ...
   END SUBROUTINE Sub
END PROGRAM
```

■

4.4.4.5.4 Passing Distributed Arguments - Summary

As it can be seen in Table 4.1 transfer of distributed arguments at procedure boundaries in most cases exhibits a *copy-in / copy-out* semantics. One reason for this is the lack of analysis information during the basic parallelization phase which enforces conservative assumptions. However, once appropriate information from later analysis phases is available in many cases copying of arguments at procedure boundaries may be avoided.

Distribution / **Shape**	*Explicit*	*Assumed*
Explicit	*copy-in / copy-out*	*copy-in / copy-out*
Assumed	*copy-in / copy-out*	*call by reference* / *pass descriptor*

Table 4.1: Transfer of distributed arguments at procedure boundaries (worst case)

4.4.4.5.5 Fortran 90 Intrinsics

Most intrinsic procedures can be implemented directly by transforming them to corresponding Fortran 90 intrinsics. The following intrinsic function classes can be implemented in this way:

- scalar bit manipulation intrinsic functions
- the argument presence inquiry function
- all the numeric elemental functions
- all mathematical elemental functions
- all the bit manipulation elemental functions

The transformation of other intrinsics must be considered individually.

Distributed data are passed to and returned from several intrinsic functions. To avoid excessive complexity and target architecture dependence some intrinsics calls can be transformed into calls to *customized runtime library procedures*. This approach has been taken, e.g., in VFCS [34] and ADAPT [94]. However, in addition to the above transformation, more preprocessing is necessary in some cases. Like the *WHERE* statement, some intrinsic functions accept a mask expression. If the mask references distributed objects, it is evaluated and the result is stored in an temporary mask array. Then the temporary mask array is passed as an argument to the corresponding runtime library procedure.

The customized runtime library procedures can be implemented as subroutines. The *result* object is passed-in as the first argument. Every array argument (or potential array argument) is followed by its descriptor. An argument that is specified as optional is introduced by a keyword. For all runtime library procedures, *result* is specified with intent *OUT*, and all other arguments with intent *IN*.

Consider the reduction function $MAXVAL$ in Example 4.4.15, where A, B, X, Y are one-dimensional distributed arrays, and C is a two-dimensional array.

Example 4.4.15: Transformation of intrinsic function references

original code

```
X = MAXVAL(A, MASK = A .EQ. B)
Y = 3 * MAXVAL(C)
```

transformed code

```
M = A .EQ. B
CALL VF_MAXVAL(X, dsc_X, A, dsc_A, MASK = M, DSC_MASK = dsc_M)
CALL VF_MAXVAL(Temp, dsc_Temp, C, dsc_C)

Y = 3 * Temp
```

■

`VF_MAXVAL` is a subroutine contained in the Vienna Fortran runtime library and implements the Fortran 90 intrinsic `MAXVAL`. The compiler will arrange that the mask array is aligned with array A and *temp* with Y.

Chapter 5

Program Analysis

In the previous section we introduced the basic parallelization scheme which transforms a Vienna Fortran 90 program into an equivalent message passing program by inserting appropriate high level communication descriptors and by masking accesses to distributed variables in order to enforce the owner computes rule. The parallel program obtained after the basic parallelization phase specifies exactly the work distribution and communication required by the input program. However, the performance of this program is likely to be very poor. One reason for that is that computation of communication sets and evaluation of owned-masks is required for each assignment statement. Furthermore, the introduction of temporary variables may result in a large memory overhead. In the following we describe several techniques to improve the performance of the parallel program and specify which analysis information is required. This chapter is structured as follows.

Sections 5.1 and 5.2 gives an overview of data flow analysis and data dependence analysis. Section 5.3 discusses distribution analysis, and in Section 5.4 we extend overlap analysis [54] which was initially developed in the context of block or generalized block distributions to block-cyclic distributions.

5.1 Flow Analysis

Flow analysis obtains information about a program by examining both the flow of control and the flow of data through a program. It aims to determine statically (at compile time) certain properties of a program that are true on any possible execution path through the program and which serve as the basis for subsequent program transformations and optimizations. Intraprocedural flow analysis is performed on the level of a single procedure and therefore worst case assumptions have to be made in the presence of procedure calls. This is not the case with interprocedural analysis which takes the whole program into account. Flow analysis can be classified into control flow analysis and data flow analysis.

5.1.1 Control Flow Analysis

Control flow analysis determines a graphical representation of the control structure of a program. The purpose of this representation is to model all possible execution pathes through a program.

Intraprocedural control flow analysis constructs for each program unit a control flow graph $G = (N, E, s)$, where the nodes N represent basic blocks in the program unit, the edges E represent transfer of control among basic blocks and s is a unique entry node. Interprocedural control flow analysis constructs the call graph which reflects the calling relationship between procedures of the program. The call graph of a program together with the control flow graph of each program unit serves as the basis for data flow analysis.

5.1.2 Data Flow Analysis

Traditional data flow analysis examines the flow of scalar variables through a program. Scalar analysis is a prerequisite for data dependence analysis and is needed to ensure the applicability of various program transformations and optimizations. Classical data flow analysis problems like live variable analysis, available expression analysis, constant propagation, or alias analysis can be solved within monotone data flow frameworks [74]. The data flow information is structured as a lattice or semi-lattice. The solution of a data flow problem is the greatest fixpoint of a system of monotone equations. Each data flow equation is parameterized by the nodes in the control flow graph and relates the solution of the data flow problem at a certain program point with the solution at immediately preceding or succeeding points. Data flow problems are solved iteratively by propagating data flow information through the flowgraph, until no new information is obtained. Interprocedural data flow analysis problems can be classified into flow-sensitive or flow-insensitive problems, depending on whether the control flow inside a procedure is taken into account or not. A detailed discussion of scalar analysis can be found in [138].

An alternative to monotone data flow analysis is based on interval analysis of reducible flow graphs. With this method the interval partitions of a flow graph are computed and a sequence of interval graphs is constructed until a trivial flow graph is obtained. Interval analysis is also referred to as structure-based flow analysis since the interval graphs represent the hierarchical structure of the flow graph. Details about this data-flow analysis technique can be found in [1]. Whereas structure based interval analysis may be more time and space efficient it is harder to implement and not as flexible and extensible as the lattice based analysis technique.

5.2 Data Dependence Analysis

The concept of data dependence enables the compiler to relax the sequential execution order of statements and thus to exploit the parallelism implicit in a program. This is especially important in the context of do loops. If dependence analysis determines that the iterations of a loop are independent, the loop can be parallelized by distributing the loop iteration space among the processors of the parallel machine. Within this thesis we do not discuss the parallelization of do loops. For more information on the parallelization of do loops see for example [51].

Data dependence information usually is represented by means of a dependence graph where the nodes represent statements and the edges represent dependences. Precise dependence information is a prerequisite for many program restructuring and loop transformation techniques [1, 2, 3, 55, 130, 131]. Data dependence analysis is discussed in detail in [138].

A major deficiency of classical data flow analysis is that references to arrays are not distinguished from references to scalars. As a consequence, if an array element or an array section is passed to a procedure, it is assumed that the whole array may be used or defined within the procedure. This limitation very often inhibits the automatic parallelization of do loops containing procedure calls. However, there exist several interprocedural analysis techniques that aim to overcome these problems by determining summary information about the array sections accessed within a procedure.

Callahan and Kennedy [28] introduced regular section descriptors (RSD) to summarize array accesses within procedures. By means of RSDs it is possible to describe a single row, a single column, a single diagonal or whole arrays. Restricted regular section descriptors and bounded regular section descriptors are an extension of the above method [63]. Triolet et.al [119] describe the set of array elements accessed within a procedure by means of so-called triolet regions. A triolet region is represented as a set of linear inequalities allowing that arbitrary convex shapes can be described. Whereas triolet regions are more general than RSDs they are more expensive to implement and use. Burke and Cytron [26] suggest to represent references to multidimensional arrays by linearizing the subscript expressions and keeping a list of linear addresses. Li and Yew [89] represent array accesses by storing the subscript expressions and loop bound variables in a structure called atom image. This method is much more precise than RSDs but introduces a large storage overhead. Data access descriptors, suggested by Balasundaram [7], provide a method for both representing the shape of array accesses as well as the traversal order. By means of data access descriptors it is possible to describe restricted convex shapes where the boundaries must be either parallel or at a 45° angle to a coordinate axis.

5.3 Distribution Analysis

The goal of distribution analysis is to determine the set of distributions that may reach a certain reference to a distributed array. Distribution analysis has to be performed on an interprocedural basis in order to avoid high overheads due to making conservative assumptions at procedure boundaries. If interprocedural distribution analysis reveals that there is only a single distribution (class) that may reach a certain array reference, more efficient code can be generated. The complexity of interprocedural distribution analysis is dependent on whether arrays may be statically or dynamically distributed.

5.3.1 Distribution Analysis for Static Distributions

We discuss first the case where only static distributions are allowed. If a program contains only statically distributed objects, then the distribution of an object is either fixed or it may be inherited from the caller. As a consequence, dummy arguments with inherited (assumed) distribution may be associated with different distributions in different incarnations of a procedure. Since the distribution of statically distributed arrays is fixed within the scope of declaration, the problem of determining which distributions may reach a dummy argument can be formulated as a flow insensitive problem. The set of distributions that may be associated with a assumed distribution dummy array can be determined by propagating distribution information in a single pass in invocation order (i.e.

top-down) over the call graph. Interprocedural distribution analysis for arrays with inherited distribution, has been implemented in the VFCS and is described in detail in [49, 51]. The output of this analysis is for each procedure a set of *partition vectors*. For each incarnation of a procedure a partition vector is computed which determines for each dummy array F the corresponding actual array (and its distribution) to which F is bound. A partition vector therefore not only contains the information about which actual arrays may be bound to a dummy array but also all possible combinations of distributions for the dummy arrays in all incarnations.

5.3.2 Distribution Analysis for Dynamic Distributions

In case of dynamically distributed objects, distribution analysis is more complex since the distribution of an array may be changed either through the execution of a distribute statement or as the result of a procedure call. The problem of determining which distributions may reach a certain array reference (reaching distribution analysis) is a flow-sensitive data-flow problem, since dynamic data distribution may be affected by control flow.

Reaching distribution analysis determines for each array reference the set of distribution specifications that may reach this reference. This can be done in two phases. During the local analysis phase the set of distribution specifications that may reach a certain array reference is determined in the same way as reaching definitions [138]. In the interprocedural propagation phase the information obtained in the local analysis phase is propagated over the call graph. Since the distribution of an array may be changed as the result of a procedure call, reaching distribution analysis can no longer be performed within a single pass over the call graph. Hall et.al [61] describe interprocedural distribution analysis for Fortran D. In Fortran D there is no distinction between static and dynamic distributions i.e. all arrays are conceptually dynamically distributed. However, the distribution of an array must not be changed as the result of a procedure call. This restriction makes it possible to perform reaching distribution analysis in a single top-down pass over the call graph.

5.4 Overlap Analysis

The goal of overlap analysis is to exploit static information about the distribution of arrays and their associated access patterns in order to produce more efficient target code. Under certain restrictions the compiler can determine a conservative approximation for the communication that is necessary for particular arrays. This information is represented by means of overlap descriptions which were introduced in Section 4.2.4.6. An overlap description of an array is a runtime independent representation for the set of non-local elements that are accessed in certain array references. Overlap descriptions are used by the compiler to statically allocate storage for non-local elements in locations adjacent to the local segment and to significantly improve the organization of communication. Moreover, they provide a suitable means for expressing the locality of computations. The overlap concept is, however, only applicable if the precise communication pattern can be determined at compile time, as it is the case for many regular computations.

The effectiveness of the overlap concept has been shown in the parallelization system SUPERB [137] in the context of block and general block distributions. In the following we briefly discuss the overlap concept as developed by Gerndt [54]. We then present how overlap descriptions can be determined for array assignment statements and show how this method can be extended to block-cyclic distributions and how it can be integrated with the more general approach based on temporary buffers.

In SUPERB overlaps are the central concept for the management of non-local data and are the basis for several program optimizations. The overlap area is the smallest contiguous area around the local segment of an array, containing all non-local elements accessed. It is represented by means of a global overlap description, which specifies the width of the overlap area with respect to the lower and upper bound in each dimension of the local segment of a distributed array.

Gerndt has developed a method for determining overlap descriptions for scalar assignments of the form $A(f_1, \ldots, f_n) = B(e_1, \ldots, e_m)$. The arrays A and B must have block or general block distributions. The assignment statement may appear in an arbitrarily nested loop. The subscript expression f_i, e_j must be linear in the loop variables.

For each processor a system of inequalities, based on the bounds of the local segments of A and the subscript expressions f_i, is constructed. By applying the Fourier-Motzkin method [39, 84] to this inequality system a set of restrictions for the loop variables is determined for each processor. By substituting these restrictions in the subscript expressions of B (i.e. e_i) a lower and upper bound for the references to B is determined for each processor. These bounds are compared to the local segment of B yielding the overlap description with respect to a particular processor. The final overlap description for an assignment statement is then determined by computing the maxima and minima over all processors. A global overlap descriptions for B is obtained by means of interprocedural overlap analysis. The global overlap description corresponds to the maximum of all statement specific overlap descriptions with respect to B. If not enough compile time information is available, conservative assumptions have to be taken. This may imply that the whole array B has to be allocated on each processor. This is for example the case if the subscript expression are dependent on run time data or the subscript expressions are non-linear. The algorithms for overlap analysis as implemented in SUPERB are described in detail in [51].

As has been shown in SUPERB, overlaps are well suited for efficiently utilizing the locality of data accesses in computations that require only nearest neighbor communication. The overlap concept has two major advantages compared to the general method of communication management based on temporary buffers.

- Uniform access to local and non-local data.

  Since the non-local elements received from other processors are stored within the local segment of the array, no temporary buffers are needed. As a consequence, splitting of local and non-local references is not necessary.

- No explicit computation of communication sets.

  The overlap area of an array specifies which local elements have to be communicated between the processors and thus at runtime explicit computation of send and receive sets can be avoided.

However, the overlap concept is not generally applicable. It can only be applied if the distribution of arrays, their access patterns, and the resulting communication can be fully determined at compile time. If there is not enough compile time information available, worst case assumptions have to be made. In general, the overlap concept can be applied to arrays only if the overlap description is known at the time the array is allocated. The overlap concept is not applicable for cyclic distributions since the local index set of an array with a cyclic distribution is always discontiguous. However, if the local index set can be described by means of a union of contiguous index sections, as it is the case for block cyclic distributions, each single index section can be extended by overlaps. This will be described in more detail in Section 5.4.2.

5.4.1 Computation of Overlap Descriptions

In the following we show how overlap descriptions for array assignments can be determined. The overlap area for a particular array A is represented by means of an overlap description, denoted by OD^A, which is determined by the maximum offsets for every dimension of the local segment of A, over all processors. For an n-dimensional array A, the overlap description is of the form

$$OD^A = [O_{l_1}/O_{r_1}, \ldots, O_{l_n}/O_{r_n}]$$

where, O_{l_i} and O_{r_i} denote the offsets with respect to the lower and upper bound of dimension i. The overlap description for an array A is determined as follows. First, an overlap description for A is determined for each assignment statement S that contains a reference to A on the right hand side. These so-called statement specific overlap descriptions are denoted by $OD^A(S)$. After that, all statement specific overlap descriptions are combined and the maximum overlap description is determined.

In the following we present two tests for determining overlap descriptions for array assignment statements. We base our assumptions on the following general form of an array assignment statement S:

$$S : A(\vec{lhs}) = B(\vec{rhs})$$

where $\vec{lhs}$ and $\vec{rhs}$ are section-subscript lists. Let furthermore A and B be distributed arrays and let P denote the set of available processors with an individual processor denoted by p, $1 \le p \le P$.

1. **General Test**

   The general test determines the overlap description with respect to a particular processor p by computing the maximum index of the receive sets for array B and comparing it to the bounds of the local index set $Local^B(p)$.

   The overlap description for array B with respect to a particular processor p, denoted by $od^B(p)$, is given by

$$od^B(B) = [o_{l_1}(p) : o_{r_1}(p), \ldots, o_{l_n}(p) : o_{r_n}(p)] \tag{5.1}$$

   where the offsets on processor p with respect to the lower and upper bound of dimension i, $o_{l_i}(p)$ and $o_{r_i}(p)$ are determined as follows:

Case 1: Block or General Block Distribution

$$
\begin{aligned}
o_{l_i}(p) &= max(0, \mathcal{L}^{B_i}(p) - min(Recv^{B_i}(p))) && (5.2) \\
o_{r_i}(p) &= max(0, max(Recv^{B_i}(p)) - \mathcal{U}^{B_i}(p)) && (5.3)
\end{aligned}
$$

where $\mathcal{L}^{B_i}(p)$ and $\mathcal{U}^{B_i}(p)$ denote the lower bound and upper bound of $Local^{B_i}(p)$ and

$$
Recv^{B_i}(p) = \bigcup_{1 \le q \le P \wedge q \ne p} Recv^{B_i}(p, q) \qquad (5.4)
$$

Case 2: Block-Cyclic Distributions

For block-cyclic distributions the overlap offsets in dimension i on processor p are determined by examining the bounds of all non-empty index sections of the local index set. This is expressed in the following algorithm.[1]

$o_{l_i}(p) := 0;\;\; o_{l_i}(p) := 0$
for each $r \in Recv^{B_i}(p)$
  $lhelp := \infty;\;\; rhelp := \infty$
  do $j := s_p$ **to** e_p
    if $(r < \mathcal{L}_j^{B_i})$ **then**
      $lhelp := min(lhelp, \mathcal{L}_j^{B_i} - r);$ **exit**
    end if
  end do
  do $j := s_p$ **to** e_p
    if $(r < \mathcal{U}_j^{B_i})$ **exit**
    $rhelp := min(rhelp, r - \mathcal{U}_j^{B_i})$
  end do
  if $(lhelp < rhelp)$ **then**
    $o_{l_i}(p) := max(o_{l_i}(p), lhelp)$
  else
    $o_{r_i}(p) := max(o_{r_i}(p), rhelp)$
  endif
end for

Where s_p and e_p denote the section numbers of the first and last non-empty index sections, $\mathcal{L}_j^{B_i}$ and $\mathcal{U}_j^{B_i}$ denote the lower and upper bound of the $j-th$ index section. Note that in case of a direct block-cyclic distribution $s_p = 1$ and $e_p = n_p$.

The left and right overlap offset O_{l_i} and O_{r_i} is the maximum over all processors of $o_{l_i}(p)$ and $o_{r_i}(p)$, respectively.

$$
\begin{aligned}
O_{l_i} &= max_{\forall p}(o_{l_i}(p)) \\
O_{r_i} &= max_{\forall p}(o_{r_i}(p))
\end{aligned}
$$

[1] To simplify the representation, we assume that in case of alignment the alignment stride is positive.

In order to determine the overlap description by means of the general test as shown above, the following restrictions have to be fulfilled:

- The number of processors on which the program will execute has to be fixed at compile time.
- The sizes of the arrays must be compile time constants.
- The list of section-subscripts $(\vec{lhs}, \vec{rhs})$ must not depend on runtime data.

If any of these restrictions is not satisfied, the receive sets cannot be computed by the compiler. In this case, no overlap description can be determined and the assignment statement has to be handled using the general method using temporary buffers.

2. **Simple Test**

The second test for computation of overlap descriptions is more efficient than the general test since it can be applied without the need of explicitly computing the receive sets. The overlap description is determined directly by comparing the lower bounds of the corresponding *lhs* and *rhs* subscript triplets in each dimension. However, this test is only applicable if both arrays have the same shape, the same distribution and the strides of the subscript triplets in corresponding dimensions are equal. If this is the case, the overlap description can be determined by symbolic subtraction of the lower bound expressions (or the subscript expressions) in each dimension. If the result of this subtraction can be evaluated at compile time, it determines the overlap value in the corresponding dimension of array B. Otherwise the test fails and no overlap description can be determined. In this case, the array assignment statement has to be processed using the general method of buffer communication.

More exactly, assume that the assignment statement is of the form

$$A(ssa_1, \ldots, ssa_n) = B(ssb_1, \ldots, ssb_n)$$

where the section-subscripts $ssa_i, ssb_i, 1 \le i \le n$ are either subscript triplets or scalar integer expressions. If ssa_i is a subscript triplet, then the corresponding ssb_i must be a subscript triplet as well. If ssa_i is a scalar integer expression, then ssb_i must be a scalar integer expression. Assume that subscript triplets are of the form $la_i : ua_i : sa_i$ and $lb_i : ub_i : sb_i$, with $sa_i = sb_i$, and scalar integer expressions are of the form ea_i and eb_i, respectively. Let $c_i, \forall i, 1 \le i \le n$, be defined as follows:

$$c_i = \begin{cases} lb_i - la_i & \text{if } ssa_i \text{ and } ssb_i \text{ are section subscripts} \\ eb_i - ea_i & \text{if } ssa_i \text{ and } ssb_i \text{ are scalar subscripts} \end{cases}$$

If $\forall i, 1 \le i \le n$, c_i can be evaluated at compile time, then the overlap in dimension i is given by:

$$O_{l_i} = \begin{cases} 0 & \text{if } c > 0 \\ -c & \text{otherwise} \end{cases}$$

$$O_{r_i} = \begin{cases} c & \text{if } c > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that opposed to the general test the simple test can be applied successfully even if the number of processors on which the program will execute is not known at compile time.

In the next section we show that under certain restrictions the overlap concept as introduced in Section 4.2.4.6, can be applied to arrays with block-cyclic distributions. In particular, we show how overlaps can be utilized to optimize shift or stencil operations for block cyclic distributions, where each processor only communicates with its nearest neighbor processors.
We first specify the overlap concept for direct block-cyclic distributions (no alignment) and then extend the method to implicit block-cyclic distributions considering arbitrary linear alignment functions.

5.4.2 Overlap Concept for Direct Block-Cyclic Distributions

The local index set on a particular processor of an array with a block-cyclic distribution can be represented as a union of contiguous index sections. For computations that require that each processor communicates only with its direct neighbors each index section can be extended by an overlap area in order to store elements received from neighbor processors.

Example 5.4.1:

```
PROCESSORS R(P)
REAL, DISTRIBUTED(CYCLIC(M)) :: A(N)

S1:   A(l : u) =  A(l+c : u+c)
S2:   A(l : u) =  A(l : u) + A(l+c : u+c) + A(l-c : u-c)
```

∎

Assume the code fragment of example 5.4.1. As shown in Section 4.2.1.1 the local index set of A on a particular processor p can be represented by a union of contiguous index sections σ_j^A, $1 \leq j \leq n_p$. The size of each index section (possibly except the last one) is M.

Consider first the assignment statement S1. It has the effect that the array section $A(l+c : u+c)$ is shifted c positions to the left if $c > 0$, and $-c$ positions to the right if $c < 0$. The overlap description with respect to statement S1 can be determined symbolically by applying the simple test yielding $OD^A(S1) = [0/c]$ if $c > 0$ and $OD^A(S1) = [-c/0]$ if $c < 0$.

If $abs(c) < M$ is satisfied, then each processor has to communicate at most with its left and/or right neighbor processor, respectively. If the value of c is known at allocation time of array A, then additional memory can be allocated to hold the elements that have to be received from the neighboring processors.

More exactly, if $0 < c < M$ then each processor p has to send the first c elements of each index section to its left neighbor processor $left(p)$. Similarly, if $-M < c < 0$ then each processor has to send the last $-c$ elements of each index section to its right neighbor $right(p)$. Therefore, on all processors the size of each index section has to be increased by c in order to store the non-local elements that are received from the neighbor processor.

For statement S2 the simple test yields an overlap description $OD^A(S2) = [c/c]$ if $c > 0$ and $OD^A(S2) = [-c/-c]$ if $c < 0$. The global overlap description for the above code fragment for A is given by $OD^A = OD^A(S2)$. As a consequence, the size of the local index set of A on each processor is then given by $(M + 2 * abs(c)) * n_p$. This is illustrated in Figure 5.1 assuming that $c = 1$.

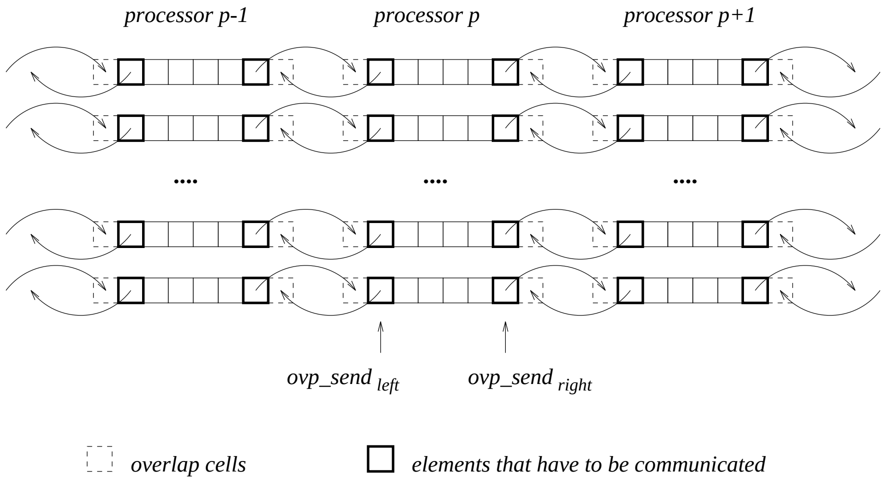


Figure 5.1: Overlap communication for block-cyclic distributions

In the following we take a closer look at the communication that is necessary for boundary processors, i.e processor 1 and processor P. Assume first that array A in the above example has a block distribution. In this case, an interior processors p $(1 < p < P)$ has to communicate with both its left and right neighbor processors $p-1$ and $p+1$, whereas processor 1, for example, only has to communicate with processor 2. In case of block-cyclic distributions, the communication behavior is slightly different since, the local index set usually consists of more than one index section (i.e $n_p > 1$). As a consequence, processor 1 also has to communicate with processor P and vice versa. The left and right neighbors of a particular processor p therefore can be determined by the following functions:

$$left(p) = ((p-1)-1)\ mod\ P + 1 = (p-2)\ mod\ P + 1 \tag{5.5}$$
$$right(p) = ((p-1)+1)\ mod\ P + 1 = (p\ mod\ P) + 1 \tag{5.6}$$

In order to guarantee that during the execution of an assignment statement based on the overlap concept each processor only communicates with at most its left and right neighbor processor, the left and right extent in the overlap description have to be less than the block size of the distribution in the corresponding dimension. This is stated in the following theorem.

Theorem 5.4.1:

> *Consider an array $A(N)$ with a $CYCLIC(M)$ distribution with respect to P processors, and an overlap description $OD^A = [O_l/O_r]$. In order to update the overlap area according to OD^A, each processor p, $1 \le p \le P$, has to communicate only with its direct left and right neighbor processors, left(p) and right(p), iff $0 < O_l, O_r \le M$.* ■

The set of elements of an array A a processor p has to send to its direct neighbor processors is called overlap send set and denoted by $Ovp\_Send^A(p)$. It consists of two subsets, those elements that have to be sent to its left neighbor and those that have to be sent to its right neighbor.

$$Ovp\_Send^A(p) = ovp\_send^A_{left(p)}(p) \cup ovp\_send^A_{right(p)}(p)$$

Before we derive formulas for the computation of these sets we discuss the implications of the overlap concept with respect to the memory allocation and index conversion methods.

Memory Allocation and Index Conversion

Assume that for an array A with a block-cyclic distribution the overlap description $OD^A = [O_l/O_r]$ has been determined. Since each index section of the local index set has to be extended by O_l to the left and O_r to the right, the size of the local index set on a particular processor including the overlap is given by

$$\widetilde{Size}(local^A(p)) = |local^A(p)| + (O_l + O_R) * n_p \tag{5.7}$$

where n_p denotes the maximum number of index sections on processor p. (See Formula 4.9.)

As a consequence of this allocation scheme, the index conversion function as specified in Formula 4.63 can no longer be applied. The function for converting global indices to local indices in the case where the local segment has been extended according to the above overlap description is denoted by $\widetilde{g2l}(i)$ and defined as follows:

$$\widetilde{g2l}(i) \quad = \quad \tilde{o}(i) + \tilde{M} * n(i) + 1 \tag{5.8}$$

where $\tilde{M} = M + O_l + O_r$, and $\tilde{o}(i) = o(i) + O_l$.

Computation of Overlap Areas

The overlap area of an array A on a particular processor p ist denoted by $Overlap^A(p)$ and can be split into the left and right overlap area, denoted by $ovp^A_{left(p)}(p)$ and $ovp^A_{right(p)}(p)$.

$$Overlap^A(p) = ovp^A_{left(p)}(p) \cup ovp^A_{right(p)}(p)$$

The left and right overlap area can be described (using local indices) by means of O_l or O_r sections, respectively, as follows[2]:

[2]We assume that arrays are allocated with lower bound 1.

$\forall p, 1 \leq p \leq P$

$$ovp^A_{left(p)}(p) = \begin{cases} \bigcup_{i=1}^{O_l} [\tilde{M} + i : ol\_ub(p) : \tilde{M}] & \text{iff } p = 1 \\ \bigcup_{i=1}^{O_l} [i : ol\_ub(p) : \tilde{M}] & \text{otherwise} \end{cases} \tag{5.9}$$

$$ovp^A_{right(p)}(p) = \bigcup_{i=1}^{O_r} [\tilde{M} - O_r + i : or\_ub(p) : \tilde{M}] \tag{5.10}$$

where

$$ol\_ub(p) = \widetilde{Size}(local^A(p))$$
$$or\_ub(p) = \widetilde{Size}(local^A(p)) - max(0, O_r - N - i\_max(p))$$

and $i\_max(p)$ is the (global) index of the last element of A on processor p.

Computation of Overlap-Send Sets

The set of elements a processor p has to send to its left and right neighbor, $ovp\_send^A_{left(p)}(p)$ and $ovp\_send^A_{right(p)}(p)$ can be represented in a similar way to the overlap areas as follows.

$\forall p, 1 \leq p \leq P$

$$ovp\_send^A_{left(p)}(p) = \begin{cases} \bigcup_{i=1}^{O_r} [\tilde{M} + O_r + i : sl\_ub(p) : \tilde{M}] & \text{iff } p = 1 \\ \bigcup_{i=1}^{O_r} [O_r + i : sl\_ub(p) : \tilde{M}] & \text{iff } p > 1 \end{cases} \tag{5.11}$$

$$ovp\_send^A_{right(p)}(p) = \bigcup_{i=1}^{O_l} [\tilde{M} - O_l - O_r + i : sr\_ub(p) : \tilde{M}] \tag{5.12}$$

where

$$sl\_ub(p) = \widetilde{Size}(local^A(p)) - O_r$$
$$sr\_ub(p) = \begin{cases} \widetilde{Size}(local^A(p)) - O_r - O_l & \text{if } p = p\_last \\ \widetilde{Size}(local^A(p)) & \text{otherwise} \end{cases}$$

Since the communication behavior of the boundary processors is different compared to that of interior processors, the left overlap area of the first processor is represented by a separate formula. The first element that has to be received from the left neighbor processor (i.e processor P) is needed for the computation of the first element in index section σ_2. Therefore, the first element of the left overlap area on processor 1 has the local index $\tilde{M} + 1$ (as opposed to index 1 on the other processors). An upper bound for the right overlap area usually is given by $\widetilde{Size}(local^A(p))$.

However, if in the last index section on the right neighbor of a particular processor p not at least O_r elements are available the upper bound of the right overlap area on processor p has to be corrected by subtracting $max(0, O_r - N - i\_max(p))$.

Analogous boundary conditions are encountered for the set of elements that have to be sent by the first processor to its left neighbor (i.e. the last processor), and for the set of elements that have to be sent by the last processor to its right neighbor (i.e. processor 1).

The execution of an array assignment statement based on the overlap concept is then done in two steps.

1. Update overlap area
   The left and right overlap area (if any) is updated on each processor by exchanging messages with the left and right neighbor processor, respectively.

2. Execute assignment to local array section
   The assignment is executed locally on each processor based on information contained in the processor's work set.

In order to update the left overlap area on a particular processor p, processor $q = left(p)$ has to send the elements designated by $ovp\_send^A_{right(q)}(q)$ to processor p.

$$ovp^A_{left(p)}(p) \leftarrow ovp\_send^A_{right(q)}(q), \text{ where } q = left(p)$$

Similarly, in order to update the right overlap area on processor p processor $q = right(q)$ has to send the elements designated by $ovp\_send^A_{left(q)}(q)$ to processor p.

$$ovp^A_{right(p)}(p) \leftarrow ovp\_send^A_{left(q)}(q), \text{ where } q = right(p)$$

In Figure 5.2 the communication that occurs between the boundary processors is shown for the array assignment of Example 5.4.2.

Example 5.4.2: Overlap concept for arrays with a block-cyclic distribution

```
PROCESSORS R(5)
REAL, DISTRIBUTED(CYCLIC(6)) :: A(120)

A(2 : 120) = A(1 : 119)
```

For this example the overlap description $OD^A = [1/0]$ *is obtained. As shown in Figure 5.2 processor* 5 *sends the array section* $A(7 :: 7)$ *(which in global indices corresponds to* $A(30 : 90 : 30)$*) to processor* 1 *which assigns it to* $A(8 :: 7)$. *Note that in this example the first memory location on processor* 1 *is never used.* ∎

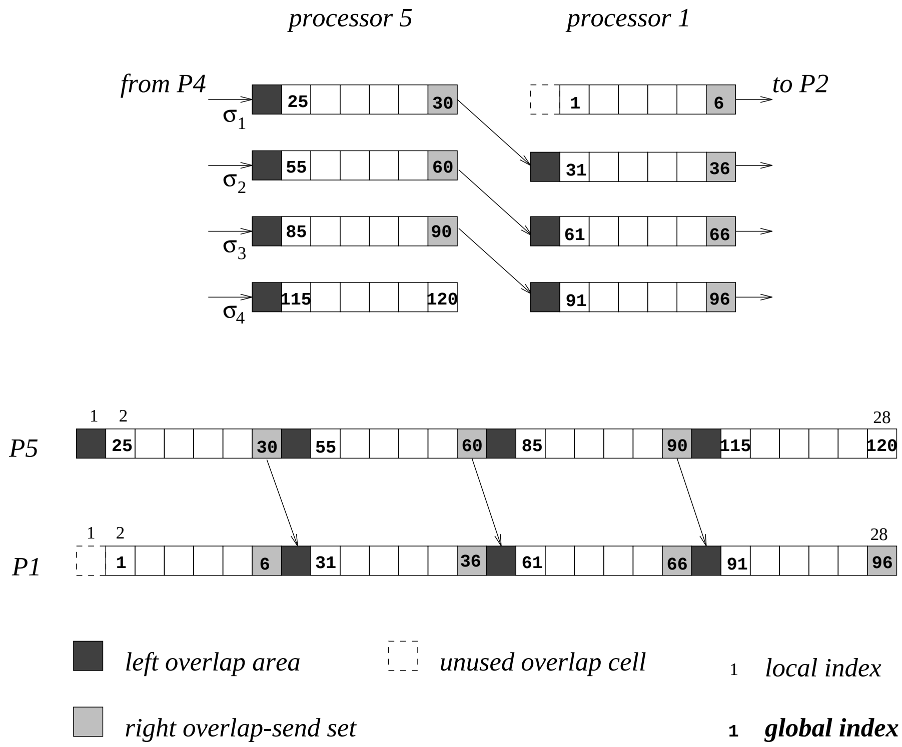


Figure 5.2: Updating overlaps in shift computations

5.4.3 Overlap Concept for Implicit Block-Cyclic Distributions

The computation of the overlap areas in case of implicit block-cyclic distributions is more complicated since the sizes of the index sections of the local index set may be different (see Section 4.2.2.6). As shown in Section 4.2.2.6 the set of elements owned by a particular processor can be described by means of the minimal part of the local index set, which consists of Σ_p indices. In the following we show how the overlap area can be described based on the minimal part of the local index set.

Example 5.4.3: Overlaps for implicit block-cyclic distributions

```
PROCESSORS R(P)
REAL, DISTRIBUTED(CYCLIC(M))     :: A(N2)
REAL, ALIGNED(B(I) WITH A(k*I+D) :: B(N)
```

■

Consider the declaration of array B in the above example.[3] Let the minimal part of the local index set $MinLoc_A^B(p)$, the section period table SP_p, the number of the first non-empty index section s_p, the number of the last non-empty index section e_p, and the offset of the first element o_p be given as defined in Section 4.2.2.6. Assume furthermore that an overlap description for B, $OD^B = [O_l/O_r]$, has been determined.

[3]For sake of simplicity of representation we assume in the following that $k > 0$.

As before, we consider the overlap concept only for array assignment statements where each processor communicates at most with its left and (or) right neighbor processor. This is the case if both the left and right extent in the overlap description is less than or equal the size of the index sections σ_i^B, $s_p < i < e_p$, (see also Theorem 1). This is stated in the following theorem.

Theorem 5.4.2:

Consider an array B that is aligned by means of an alignment function $\alpha_A^B(i) = k * i + d$ with an array A which has a $CYCLIC(M)$ distribution with respect to P processors. In order to update the overlap area according to the overlap description $OD^B = [O_l/O_r]$, each processor p, $1 \leq p \leq P$, has to communicate only with its direct left and right neighbor processors, left(p) and right(p), iff $0 < O_l, O_r \leq \left\lfloor \frac{M}{k} \right\rfloor$. ∎

Memory Allocation and Index Conversion

Memory allocation with respect to the overlap description is done in the same way as shown in the previous section by increasing the size of each index section σ_i, $s_p \leq i \leq e_p$ by $O_l + O_r$. The size of the local index set on a particular processor including the overlap area is given by

$$\widetilde{Size}(local_A^B(p)) = |local_A^B(p)| + (O_l + O_R) * (e_p - s_p + 1) \tag{5.13}$$

where according to Formula 4.9 n_p denotes the maximum number of index sections on processor p.

As a consequence, the index conversion function of Formula 4.71 has to be modified as follows.

1. The section period table has to be changed in order to reflect that the length of each index section has been increased. This is expressed in the following simple algorithm:

Algorithm 5.4.1: Modify SP-table according to an overlap description

Input: `section-period table on particular processor` p, SP_p
`left and right extent of overlap area,` O_l, O_r
Output: `modified section-period table,` $\widetilde{SP}_p$
`number of indices contained in one cycle including non-local indices stored in overlaps,` $\widetilde{\Sigma}_p$

$\widetilde{SP}_p[0] := 0$
for $i := 1$ **to** $Size(SP_p)$
    $\widetilde{SP}_p[i] := SP_p[i] + i * (O_l + O_r)$
end for
$\widetilde{\Sigma}_p := SP_p[Size(SP_p)]$

2. The relative offset $o'(i)$ has to be increased by O_l.

$$\tilde{o}(i) = o'(i) + O_l$$

The modified index conversion function which takes the overlap description into account is then given by:

$$\widetilde{g2l}(i) = \left\lfloor \frac{n'(i)}{t_p} \right\rfloor * \widetilde{\Sigma}_p + \widetilde{SP}_p[n'(i) \; mod \; t_p] + \tilde{o}(i) + 1 \quad (5.14)$$

Note that the function $\widetilde{g2l}$ cannot be used to covert the indices of elements that are in the overlap area of a particular processor p. It may only be used for elements that are owned by p.

Computation of Overlap Areas

The computation of the overlap areas for array B on a particular processor p is based on the minimal part of the local index set, $MinLoc_A^B(p)$, as introduced on page 108. This is described in algorithm 5.4.2. The algorithm determines for a particular processor p the set of those indices of the minimal part of the local index set that are in the left and right overlap area of p. These are the so-called sets of **relevant indices for the overlap areas**, denoted by OL_p and OR_p, respectively. The overlap areas then can be described by means of the indices contained in OL_p and OR_p.

In the first phase of algorithm 5.4.2 all those indices of $MinLoc_A^B(p)$ are determined that are at a boundary of an index section of the local index set of p according to the block-representation of Formula 4.37. Boundary indices are stored in the arrays $LB[1:cnt]$ and $UB[1:cnt]$, respectively. Since the local index set exhibits a periodic behavior, we have $cnt < \mathcal{C} + 1$. The array of boundary indices is constructed by simply scanning through the minimal part of the local index set (lines 1-7 in algorithm 5.4.2). Assume that $MinLoc_A^B(p) = \{l_1, \ldots, l_{\Sigma_p}\}$ is given. Obviously l_1 and l_{Σ_p} are boundary elements. An element $l_i \in MinLoc_A^B(p)$, $1 < i < \Sigma_p$, is a boundary element if $l_i - l_{i-1} > 1$.

After that, the relevant indices of the left and right overlap areas, OL_p and OR_p, are determined based on the array of boundary elements $LB[1:cnt]$ and $UB[1:cnt]$. Depending on the first index that has been mapped to a particular processor a special case might have to be considered for the computation of OL_p (lines 14-18). This is discussed in more detail in Example 5.4.4.

The left and right overlap area, $ovp^B_{left(p)}(p)$ and $ovp^B_{right(p)}(p)$, on a particular processor p according to an overlap description $OD^B = [O_l/O_r]$, can be represented based on the relevant indices for the overlap areas as unions of sections as shown below. Assume that on a particular processor p the sets of relevant indices for the overlap areas are given by

$$OL_p = \{ol_1(p), \ldots, ol_{ol\_max(p)}(p)\}$$
$$OR_p = \{or_1(p), \ldots, or_{or\_max(p)}(p)\}.$$

Then $\forall p, 1 \leq p \leq P$

$$ovp^B_{left(p)}(p) = \bigcup_{i=1}^{ol\_max(p)} [ol_i(p) : ol\_ub(p) : \widetilde{\Sigma}_p] \quad (5.15)$$

$$ovp^B_{right(p)}(p) = \bigcup_{i=1}^{or\_max(p)} [or_i(p) : or\_ub(p) : \widetilde{\Sigma}_p] \quad (5.16)$$

Algorithm 5.4.2: Relevant indices for overlap area

Input: `left and right extent of overlap area,` O_l, O_r
`minimal part of local index set` $MinLoc_A^B(p) = \{l_1, \ldots, l_{\Sigma_p}\}$
`cycle of local index set` $\mathcal{C}$ `(Theorem 4.2.2)`
`number of indices contained in one cycle including overlaps,` $\widetilde{\Sigma_p}$
`offset of first index on processor` p, o_p `(see page 112)`

Output: `set of relevant indices for` $ovp^B_{left}(p)$, `denoted by` OL_p
`set of relevant indices for` $ovp^B_{right}(p)$, `denoted by` OR_p

$LB[1] := l_1$; $cnt := 1$
for i := 1 **to** $\Sigma_p - 1$
    if $l_i - l_{i-1} > 1$ **then**
        $UB[cnt] := l_{i-1}$; $cnt := cnt + 1$; $LB[cnt] := l_i$
    end if
end for
$UB[cnt] := l_{\Sigma_p}$

`! determine relevant indices for left overlap area`
$OL_p := \phi$
for i := 1 **to** $\mathcal{C}$
    $nr := min(O_l, LB[i] - 1)$
    $start := \widetilde{g2l}(LB[i]) - nr$
    $OL_p := OL_p \cup [start : start + nr - 1 : 1]$
end for
if $LB[1] \leq O_l$ **then**
    $nr := O_l - LB[1] + 1$
    $start := \widetilde{\Sigma_p} - o_p + 1$
    $OL_p := OL_p \cup [start : start + nr - 1]$
endif

`! determine relevant indices for right overlap area`
$OR_p := \phi$
for i := 1 **to** $\mathcal{C}$
    $start := \widetilde{g2l}(UB[i]) + 1$
    $OR_p := OR_p \cup [start : start + O_r - 1 : 1]$
end for

where

$$\begin{aligned} ol\_ub(p) &= \widetilde{Size}(local_A^B(p)) \\ or\_ub(p) &= \widetilde{Size}(local_A^B(p)) - max(0, O_r - N_B - i\_max(p)) \end{aligned}$$

and $i\_max(p)$ is the (global) index of the last element of B on processor p.

Note that on a particular processor p the upper bound of the right overlap area has to be corrected by $max(0, O_r - N_B - i\_max(p))$ if less than O_r elements with indices greater than $i\_max(p)$ are on p's right neighbor. As a consequence, on the processor to which element with index N_B is mapped (i.e. p_last), an upper bound for the right overlap area is given by $\widetilde{Size}(local_A^B(p)) - O_r$. This is the case for processor 2 in Example 5.4.4.

Computation of Overlap-Send Sets

The overlap-send sets can be determined in a similar way as the overlap areas on the basis of the minimal part of the local index set, $MinLoc_A^B(p)$. This is described in algorithm 5.4.3.
In the beginning the arrays of boundary indices are constructed in exactly the same way as in algorithm 5.4.2. After that, the set of those indices of the minimal part of the local index set that have to be sent to the left neighbor and right neighbor processor are determined based on $LB[1 : cnt]$ and $UB[1 : cnt]$. These are the so-called sets of **relevant indices for the overlap-send stes**, denoted by SL_p and SR_p respectively. Special cases (lines 8, 14-17, 24-28) have to be considered on the processor to which index 1 is mapped (i.e $p = p\_first$).

Based on the sets of relevant indices for the overlap-send sets, SL_p and SR_p, the overlap-send sets for a particular processor p, $ovp\_send^B_{left(p)}(p)$ and $ovp\_send^B_{right(p)}(p)$, can be represented as unions of sections as follows: Assume that on a particular processor p the relevant indices of the overlap-send sets are given by

$$SL_p = sl_1(p), \ldots, sl_{sl\_max(p)}(p)$$
$$SR_p = sr_1(p), \ldots, sr_{sr\_max(p)}(p).$$

Then $\forall p, 1 \leq p \leq P$

$$ovp\_send^B_{left(p)}(p) = \bigcup_{i=1}^{sl\_max(p)} [sl_i(p) : sl\_ub(p) : \widetilde{\Sigma}_p] \tag{5.17}$$

$$ovp\_send^B_{right(p)}(p) = \bigcup_{i=1}^{sr\_max(p)} [sr_i(p) : sr\_ub(p) : \widetilde{\Sigma}_p] \tag{5.18}$$

where

$$\begin{aligned} sl\_ub(p) &= Size(local_A^B(p)) - O_r \\ sr\_ub(p) &= \begin{cases} Size(local_A^B(p)) - O_r - O_l & \text{if } p = p\_last \\ Size(local_A^B(p)) & \text{otherwise} \end{cases} \end{aligned}$$

Note that upper bounds for the overlap areas and for the overlap-send sets are determined in the same way as in the previous section for direct block-cyclic distributions.

Algorithm 5.4.3: Relevant indices for overlap-send sets

Input: `left and right extent of overlap area,` O_l, O_r
`minimal part of local index set` $MinLoc_A^B(p) = \{l_1, \dots, l_{\Sigma_p}\}$
`cycle of local index set` $\mathcal{C}$ `(Theorem 4.2.2)`
`number of indices contained in one cycle including overlaps,` $\widetilde{\Sigma_p}$
`offset of first index on processor` p`,` o_p `(see page 112)`

Output: `set of relevant indices for` $ovp\_send_{left(p)}(p)$`, denoted by` SL_p
`set of relevant indices for` $ovp\_send_{right(p)}(p)$`, denoted by` SR_p

$LB[1] := l_1$; $cnt := 1$
for i := 1 **to** $\Sigma_p - 1$
    if $l_i - l_{i-1} > 1$ **then**
        $UB[cnt] := l_{i-1}$; $cnt := cnt + 1$; $LB[cnt] := l_i$
    end if
end for
$UB[cnt] := l_{\Sigma_p}$

`! determine elements that have to be sent to left neighbor`
if $p = p\_first$ **then** $i\_min := 2$ **else** $i\_min := 1$ **end if**
$SL_p := \phi$
for i := i_min **to** $\mathcal{C}$
    $start := \widetilde{g2l}(LB[i])$
    $SL_p := SL_p \cup [start : start + O_r - 1 : 1]$
end for
if $p = p\_first$ **then**
    $start := \widetilde{g2l}(LB[1]) + \widetilde{\Sigma_p} - o_p$
    $SL_p := SL_p \cup [start : start + O_r - 1 : 1]$
end if

`! determine elements that have to be sent to right neighbor`
$SR_p := \phi$
for i := 1 **to** $\mathcal{C}$
    $nr := min(UB[i] - LB[i] + 1, O_l)$
    $start := \widetilde{g2l}(max(LB[i], UB[i] - O_l + 1))$
    $SR_p := SR_p \cup [start : start + nr - 1 : 1]$
end for
if $p = p\_first$ **then**
    $nr := O_l - UB[1]$
    $start := \widetilde{g2l}(UB[1]) + \widetilde{\Sigma_p} - O_l + 1$
    $SR_p := SR_p \cup [start : start + nr - 1 : 1]$
end if

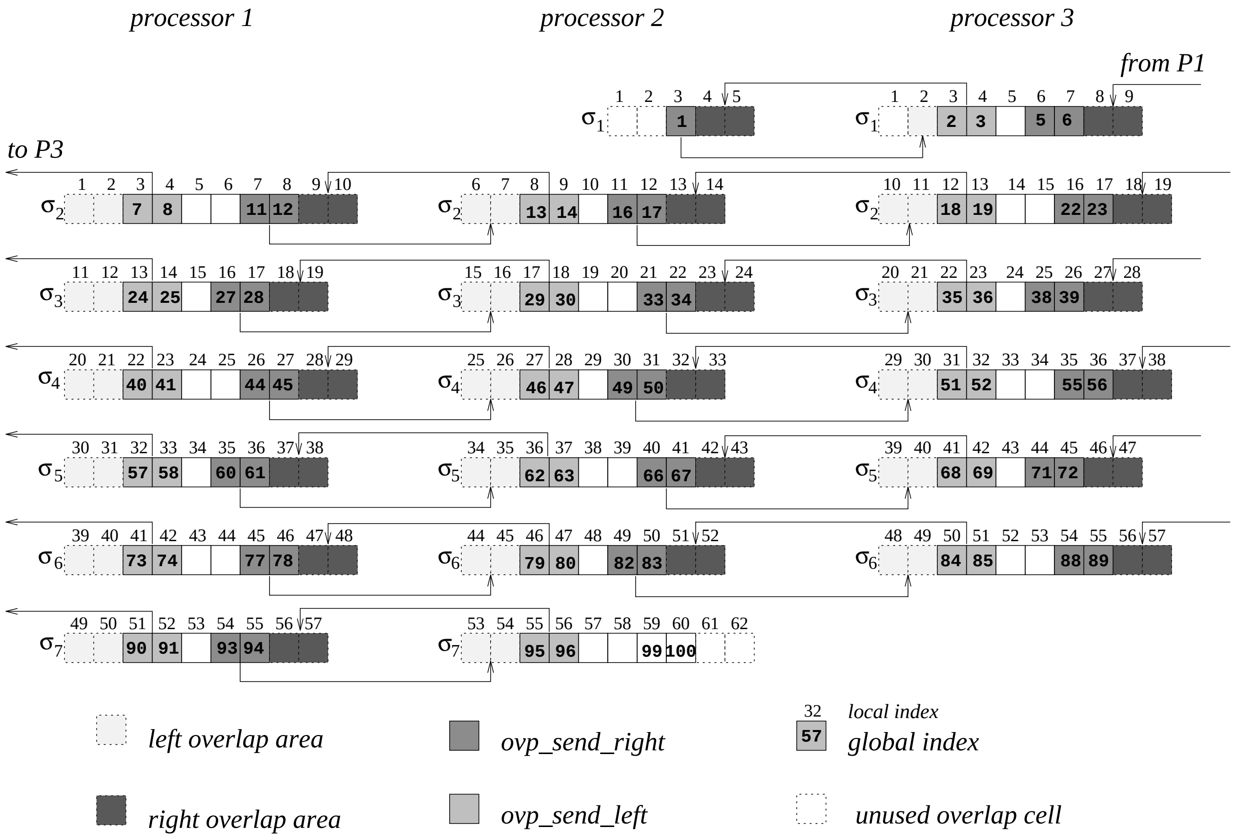


Figure 5.3: Overlap communication for implicit block-cyclic distributions

Example 5.4.4: Overlap concept for arrays with an implicit block-cyclic distribution

```
PROCESSORS R(3)

REAL, DISTRIBUTED(CYCLIC(11))      :: A(250)
REAL, ALIGNED(B(I) WITH A(2*I+20)) :: B(100)

A(L : U : S) = A(L+2 : U+2: S) + A(L-2 : U-2 : S)
```

This is an example to illustrate the algorithms for the computation of the overlap areas and the overlap-send sets with respect to the above code fragment (see also Figure 5.3).

The input to both algorithms is as follows:

$OD^B = [2/2]$
$MinLoc_A^B(1) = \{7-12, 24-18\}$, $MinLoc_A^B(2) = \{1, 13-17, 29-33\}$, $MinLoc_A^B(3) = \{2-6, 18-23\}$, $\Sigma_1 = \Sigma_2 = \Sigma_3 = 11$, $\widetilde{\Sigma}_1 = \widetilde{\Sigma}_2 = \widetilde{\Sigma}_3 = 19$, $o_1 = o_3 = 0$, $o_2 = 5$, $\mathcal{C} = 2$.

After execution of lines 1-7 the boundary indices are given by:

$LB_1 = \{7, 24\}$, $UB_1 = \{12, 28\}$
$LB_2 = \{1, 13, 29\}$, $UB_2 = \{1, 17, 29\}$
$LB_3 = \{2, 18\}$, $UB_3 = \{6, 23\}$

Overlap Areas

The left overlap area is determined according to algorithm 5.4.2. After execution of the loop 8-13 the computation of OL_1 is finished. A special case is encountered on processors 2 and 3 since on both processors $LB[1] < 2$. Therefore, the indices 15 and 16 are added to OL_2 and index 20 is added to OL_3. The relevant indices for the left overlap area finally are as shown in column 3 of the table below. The relevant indices for the right overlap area are shown in the rightmost column.

Processor	OL_p *(lines 8-13)*	OL_p	OR_p
1	$\{1, 2, 11, 12\}$	$\{1, 2, 11, 12\}$	$\{9, 10, 18, 19\}$
2	$\{6, 7\}$	$\{6, 7, 15, 16\}$	$\{4, 5, 13, 14\}$
3	$\{2, 10, 11\}$	$\{2, 10, 11, 20\}$	$\{8, 9, 18, 19\}$

Overlap-Send Sets

The overlap-send sets are computed according to algorithm 5.4.3. After execution of lines 8-13 the computation of SL_1 and SL_3 is finished, whereas on processor $p\_first = 2$ a special case is encountered. Therefore, indices 17, 18 are added to SL_2, finally yielding the left overlap-send set as shown in column 3 of the table below.
After execution of lines 18-23 computation of the right overlap-send set is finished on processors 1 and 3 (column 4 in table below). Again a special case (lines 24-28) has to be considered on processor 2 which results in adding index 21 to SR_2.

Processor	SL_p *(lines 8-13)*	SL_p	SR_p *(lines 18-23)*	SR_p
1	$\{3, 4, 13, 14\}$	$\{3, 4, 13, 14\}$	$\{7, 8, 16, 17\}$	$\{7, 8, 16, 17\}$
2	$\{8, 9\}$	$\{8, 9, 17, 18\}$	$\{3, 11, 12\}$	$\{3, 11, 12, 21\}$
3	$\{3, 4, 12, 13\}$	$\{3, 4, 12, 13\}$	$\{6, 7, 16, 17\}$	$\{6, 7, 16, 17\}$

The overlap areas and the overlap-send sets are therefore as follows:

$$ovp^B_{left(1)}(1) = [1:57:19] \cup [2:57:19] \cup [11:57:19] \cup [12:57:19]$$
$$ovp^B_{left(3)}(2) = [6:62:19] \cup [7:62:19] \cup [15:62:19] \cup [16:62:19]$$
$$ovp^B_{left(3)}(3) = [2:57:19] \cup [10:57:19] \cup [11:57:19] \cup [20:57:19]$$

$$ovp^B_{right(2)}(1) = [9:57:19] \cup [10:57:19] \cup [18:57:19] \cup [19:57:19]$$
$$ovp^B_{right(2)}(2) = [4:60:19] \cup [5:60:19] \cup [13:60:19] \cup [14:60:19]$$
$$ovp^B_{right(3)}(3) = [8:57:19] \cup [9:57:19] \cup [18:57:19] \cup [19:57:19]$$

$$ovp\_send^B_{left(1)}(1) = [3:57:19] \cup [4:57:19] \cup [13:57:19] \cup [14:57:19]$$
$$ovp\_send^B_{left(2)}(2) = [8:62:19] \cup [9:62:19] \cup [17:62:19] \cup [18:62:19]$$
$$ovp\_send^B_{left(3)}(3) = [3:57:19] \cup [4:57:19] \cup [12:57:19] \cup [13:57:19]$$

$$ovp\_send^B_{right(1)}(1) = [7:57:19] \cup [8:57:19] \cup [16:57:19] \cup [17:57:19]$$
$$ovp\_send^B_{right(2)}(2) = [3:58:19] \cup [11:58:19] \cup [12:58:19] \cup [21:58:19]$$
$$ovp\_send^B_{right(3)}(3) = [6:57:19] \cup [7:57:19] \cup [16:57:19] \cup [17:57:19]$$

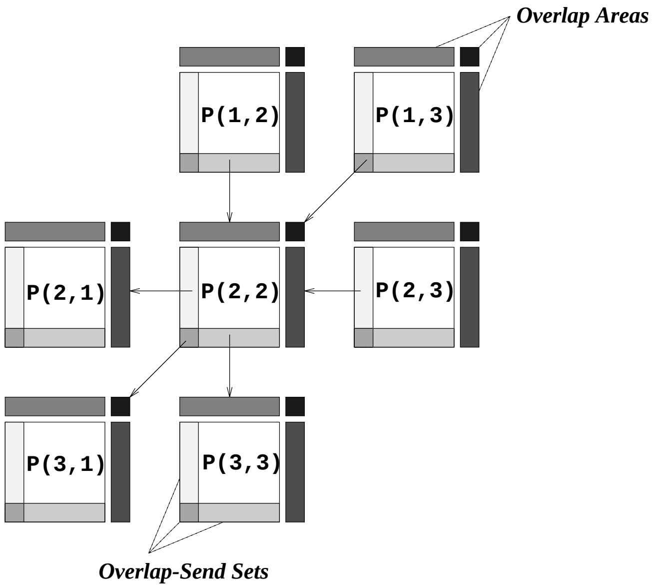
∎

Multidimensional Overlaps

Let in the following A denote a n-dimensional array that is distributed with respect to a n-dimensional processor array R and let $p = (p_1, \ldots, p_n)$ and $q = (q_1, \ldots, q_n)$ denote individual processors of R. Let dimension i of A be distributed by $CYCLIC(M_i)$ with respect to dimension i of R, $1 \leq i \leq n$. Assume that a global overlap description $OD^A = [O_{l_1}/O_{r_1}, \ldots, O_{l_n}/O_{r_n}]$.

In the case where $0 < O_{l_i}, O_{r_i} \leq M_i, \forall i, 1 \leq i \leq n$, the overlap area $Overlap^A(p)$ on a particular processor p consists of $3^n - 1$ subsets which have to be updated by $3^n - 1$ different direct neighbor processors.

Example 5.4.5: Overlap neighbors



Assume that a processor array $R(3,3)$ *and an overlap description* $[1/0, 0/1]$ *for an array with block-cyclic distribution is given. The set of processors from which processor R(2,2) might have to receive a message is given by* $NBR((2,2), [1/0, 0/1]) = \{(1,2), (1,3), (2,3)\}$. *The set of processors to which processor R(2,2) might have to send a message is given by* $NBS((2,2), [1/0, 0/1]) = \{(2,1), (3,1), (3,2)\}$.

∎

In order to determine the overlap area and the overlap send set with respect to a given overlap description OD we need to specify for a particular processor p the sets of neighbor processors to which p must send and from which p must receive. These sets are denoted by $NBS(p)$ and $NBR(p)$ and can be defined as follows:

Set of Receiving Processors:

$$NBR(p, OD) = \{q \mid \forall i, 1 \leq i \leq n, q_i \in \{nbr_i(p_i) \cup \{p_i\}\} \wedge \exists i \text{ s.t. } q_i \neq p_i\} \tag{5.19}$$

with the set of receiving neighbors with respect to dimension i, $nbr_i(p_i)$, defined as

$$nbr_i(p_i) = \begin{cases} \{left(p_i)\} & \text{if } O_{l_i} > 0 \wedge O_{r_i} = 0 \\ \{right(p_i)\} & \text{if } O_{l_i} = 0 \wedge O_{r_i} > 0 \\ \{left(p_i), right(p_i)\} & \text{if } O_{l_i} > 0 \wedge O_{r_i} > 0 \\ \phi & \text{if } O_{l_i} = 0 \wedge O_{r_i} = 0 \end{cases}$$

Set of Sending Processors:

$$NBS(p, OD) = \{q \mid \forall i, 1 \leq i \leq n, q_i \in \{nbs_i(p_i) \cup \{p_i\}\} \wedge \exists i \text{ s.t. } q_i \neq p_i\} \tag{5.20}$$

and the set of sending neighbors with respect to dimension i, $nbs_i(p_i)$, is defined as

$$nbs_i(p_i) = \begin{cases} \{right(p_i)\} & \text{if } O_{l_i} > 0 \wedge O_{r_i} = 0 \\ \{left(p_i)\} & \text{if } O_{l_i} = 0 \wedge O_{r_i} > 0 \\ \{right(p_i), left(p_i)\} & \text{if } O_{l_i} > 0 \wedge O_{r_i} > 0 \\ \phi & \text{if } O_{l_i} = 0 \wedge O_{r_i} = 0 \end{cases}$$

The overlap area on a particular processor $p = (p_1, \ldots, p_n)$ according to OD^A is given by:

$$Overlap^A(p) = \bigcup_{q \in NBR(p, OD^A)} ovp_q^A(p) \tag{5.21}$$

where

$$ovp_q^A(p) = O_{q_1}^{A_1}(p_1) \times \ldots \times O_{q_n}^{A_n}(p_n)$$

$$\forall i, 1 \leq i \leq n, O_{q_i}^{A_i}(p_i) = \begin{cases} ovp_{q_i}^{A_i}(p_i) & \text{if } q_i \neq p_i \\ loc\_local_i^A(p_i) & \text{otherwise} \end{cases}$$

and $loc\_local_i^A(p_i)$ denotes the local index set in local indices.

The overlap send sets are constructed in a similar way.

$$Overlap\_Send^A(p) = \bigcup_{q \in NBS(p, OD^A)} ovp\_send_q^A(p) \tag{5.22}$$

$$ovp\_send_q^A(p) = S_{q_1}^{A_1}(p_1) \times \ldots \times S_{q_n}^{A_n}(p_n)$$

where

$$\forall i, 1 \leq i \leq n, S_{q_i}^{A_i}(p_i) = \begin{cases} ovp\_send_{q_i}^{A_i}(p_i) & \text{if } q_i \neq p_i \\ loc\_local_i^A(p_i) & \text{otherwise} \end{cases}$$

In order to update a particular overlap area $ovp_q^A(p)$ on processor p, processor $q \in NBR(p)$ has to send the elements designated by $ovp\_send_p^A(q)$ to p.

$$ovp_q^A(p) \leftarrow ovp\_send_p^A(q)$$

Example 5.4.6 is an illustration of a 2-dimensional overlap area.

Example 5.4.6: Multidimensional overlaps

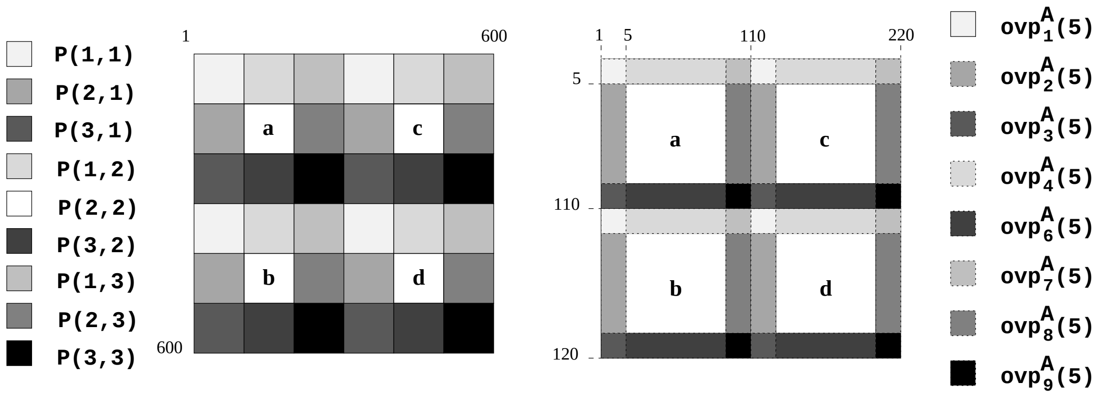


Fig. a: Distribution Pattern of A

Fig. b: Overlap Areas on Processor P(2,2)

Figure 5.4: Overlap area of a 2-dimensional array

```
PROCESSORS R(3,3)
REAL, DISTRIBUTED(CYCLIC(100),CYCLIC(100)) :: A(600)
```

In Figure 5.4.b the local segment of A together with the overlap area is shown for processor $P5 = P(2,2)$. There are 8 different overlap areas each of which is updated by a different neighbor processor. It is assumed that the global overlap description is given by $OD^A = [5/5, 5/5]$. For example on processor 5 the overlap area with respect to processor 4 is given by $ovp_4^A(5) = ([1:5] \cup [111:115]) \times ([6:105] \cup [116:215])$. ■

Chapter 6

Optimization and Code Generation

6.1 Program Optimization

In the basic parallelization scheme all references to distributed arrays are transformed in such a way that potential communication is managed via temporary buffers. For example, array assignment statements are transformed by inserting a high level communication descriptor prior to the assignment statement, and masking the assignment in order to enforce the owner computes rule. In the code generation phase communication descriptors are expanded into calls to appropriate message passing primitives which communicate the required data between processors using temporary buffers. Array assignment statements are split into two WHERE statements. One WHERE statement represents all the instances of an array assignment that can be executed without access to non-local data whereas the second WHERE statement represents all accesses that require non-local data which is stored in temporary buffers. If the code generation would take place immediately after basic parallelization, then the resulting program usually would be very inefficient for several reasons.

- There may be communication descriptors which do not result in communication and therefore can be eliminated. This however requires precise information about the set of distributions that may reach a certain array reference.

- Communication descriptors for scalar references inside of loops generate a large number of small messages and therefore should be extracted from loops and replaced by a small number of large messages.

- Parallel execution of each assignment statement requires the computation of communication sets and work sets. The preprocessing overhead for computation of these sets can be reduced by grouping statements with the same or similar communication patterns together and reusing communication sets and work sets wherever possible. Moreover, if enough compile time information is available explicit computation of these sets at runtime can be avoided by generating a parameterized representation.

- The large storage overhead introduced during the basic parallelization phase has to be reduced by allocating minimal storage for distributed arrays and reusing temporary buffers efficiently.

- Point to point communication should be replaced by more efficient collective communication primitives like broadcast or multicast.
- In many cases statement masks can be eliminated by adapting the reference space of array assignment statements and/or the loop iteration space accordingly.

Communication overhead in distributed memory multiprocessors can be characterized by two parameters: *latency* and *bandwith*. Communication latency, i.e the time between the start and the completion of a message, consists of the startup-time and the message transmission time. On most distributed memory architectures the startup time is usually one or two orders of magnitude higher than the time required for transmitting one byte.[1] It is therefore crucial to reduce the overhead imposed by the startup time by

- reducing the number of messages
- avoiding single element messages
- hiding the communication overhead by overlapping communication and computation

In the following we discuss several optimizations techniques based on high level communication descriptors that aim to reduce the communication overhead of the parallel program. In particular we focus on communication vectorization, elimination of communication descriptors, fusion of communication descriptors, reuse of communication patterns, overlapping of communication and computation, and overlap communication. The optimization methods presented are discussed in the context of array assignment statements. However, many of the concepts can also be applied to loops.

6.1.1 Communication Vectorization

Communication vectorization is an important optimization that aims to reduce the communication overhead by combining element messages into vector messages. Communication vectorization uses the results of data dependence analysis in order to determine if communication statements may be extracted from loops. As a consequence, a large number of small messages can be replaced by a single large message, reducing both latency and startup time. A communication descriptor for a particular array reference can be extracted legally from surrounding loops as far as no true dependencies are violated. Gerndt [51] has developed an algorithm that computes the appropriate loop level to insert communication descriptors.

Consider a loop nest of depth n and a communication descriptor $COMM_{ref}$ that has been generated with respect to a statement S containing a reading reference ref to a distributed variable within the innermost loop (i.e the loop with level n). The *innermost relevant loop* [54] for ref is defined as the loop with the maximum level r, $1 \leq r \leq n$,

- if there exists a statement S' and there is a loop-carried true dependence at level r of S on S'
- or there exists a statement S' at nesting level r and there is a loop independent true dependence of S on S'.

[1] On the Intel IPSC/860 the startup time is approximately $70\mu sec$ whereas the transfer of a floating point number (8 bytes) between neighbor processors takes only $1\mu sec$ once the communication has been set up.

The communication descriptor $COMM_{ref}$ then can be extracted from the $n - r$ innermost loops. If no innermost relevant loop can be determined, then $r = 0$, and the communication descriptor can be moved in front of the loop nest.

When a communication descriptor is extracted from a loop references to index variables in the access patterns of the communication descriptors have to be replaced by corresponding sections which are generated by substituting the loop bounds (and increment). If for example a communication descriptor with source access pattern $< i >$ is extracted form a loop with header `DO i=1,N,S`, then the source access pattern is replaced by $< 1 : N : S >$. Communication vectorization is shown in Figure 6.1.1.

6.1.2 Elimination of Communication Descriptors

During the basic parallelization phase communication descriptors are inserted for every reference to a distributed object appearing on the *rhs* of an assignment statement and in the case of call statements if either the actual or the corresponding formal argument is distributed. If it can be determined at compile time that a communication descriptor never generates communication, then it can be eliminated. In order to decide if a communication descriptor can be eliminated, it is necessary to know which distributions may reach the corresponding array references.

6.1.2.1 Useless Communication Descriptors

A communication descriptor is useless if it does not generate communication in all execution instances. Consider a communication descriptor of the general form $\mathbf{COMM}(\Delta^A, < l\vec{hs} >, \Delta^B, < r\vec{hs} >, tmp)$ and let $\mathcal{F} : \bar{\mathbf{I}}^{A(l\vec{hs})} \rightarrow \bar{\mathbf{I}}^{B(r\vec{hs})}$ denote the total function that maps the indices of the *lhs* array section to the corresponding indices of the *rhs* array section. The above communication descriptor does not generate communication iff

$$\delta^A(\mathbf{i}) = \delta^B(\mathcal{F}(\mathbf{i})),\ \forall \mathbf{i} \in \bar{\mathbf{I}}^{A(l\vec{hs})}.$$

In order to guarantee that the above condition is satisfied, it is usually necessary that the distribution of A and B as well as the section subscripts can be evaluated at compile time. However, there are often situations where by means of symbolic analysis it can be determined that a communication descriptor never generates communication.

Consider for example the array assignment $A(la : ua : sa) = B(lb : ub : sb)$. The corresponding communication descriptor $COMM(\Delta^A, < la : ua : sa >, \Delta^B, < lb : ub : sb >, tmp)$ can be eliminated

- if A and B have the same distribution and the *lhs* reference space equals the *rhs* reference space, i.e. $\delta^A = \delta^B \wedge la = lb \wedge sa = sb$, or
- A is aligned with B and $\alpha^A_B(i) = \left\lfloor \frac{(i-la)*sb}{sa} \right\rfloor + lb, \forall i \in \mathbf{I}^A$, or
- B is aligned with A and $\alpha^B_A(i) = \left\lfloor \frac{(i-lb)*sa}{sb} \right\rfloor + la, \forall i \in \mathbf{I}^B$.

Example 6.1.1: Communication vectorization

original code

```
PROCESSORS R(NP1,NP1)
PARAMETER (N=100)
REAL, DISTRIBUTED($\delta^A$) ::  A(N,N)
REAL, DISTRIBUTED($\delta^B$) ::  B(N)
...
   DO k=1,times
s1    B(k) = B(k+1)
      DO I=1,N
s2       A(i,1:N) = (A(i-1,1:N) + A(i+1,1:N))*B(k)
      END DO
   END DO
```

code after basic parallelization

```
   DO k=1,times
      COMM$_1$($\Delta^B, < k >, \Delta^B, < k+1 >, tmp1$)
s1    B(k) = B(k+1)
      DO I=1,N
         COMM$_2$($\Delta^A, < [i, 1:N] >, \Delta^A, < [i-1, 1:N] >, tmp2$)
         COMM$_3$($\Delta^A, < [i, 1:N] >, \Delta^A, < [i+1, 1:N] >, tmp3$)
         COMM$_4$($\Delta^A, < [i, 1:N] >, \Delta^B, < k >, tmp4$)
s2       Mask $\rightarrow$ A(i,1:N) = (A(i-1,1:N) + A(i+1,1:N))*B(k)
      END DO
   END DO
```

code after communication vectorization

```
   COMM$_1$($\Delta^C, < [1 : times] >, \Delta^C, < [2 : times+1] >, tmp1$)
   DO k=1,times
s1    B(k) = B(k+1)
      COMM$_3$($\Delta^A, < [i, 1:N] >, \Delta^A, < [i+1, 1:N] >, tmp3$)
      COMM$_4$($\Delta^A, < [i, 1:N] >, \Delta^B, < k >, tmp4$)
      DO I=1,N
         COMM$_2$($\Delta^A, < [i, 1:N] >, \Delta^A, < [i-1, 1:N] >, tmp2$)
s2       Mask $\rightarrow$ A(i,1:N) = (A(i-1,1:N) + A(i+1,1:N))*B(k)
      END DO
END DO
```

In this example the communication descriptor $COMM_1$ can be extracted from the k-loop since there are no true dependences for $s1$ caused by the reference $B(k+1)$. $COMM_2$ cannot be extracted from the i-loop because there is a loop carried true dependence at level 2 caused by the reference $A(i-1, 1:N)$. $COMM_3$ can be extracted from the innermost loop but not from the k-loop, since there is a loop carried true dependence with level 1 caused by the reference $A(i+1, 1:N)$. $COMM_4$ is inserted immediately before the $i-$loop because there is a loop-independent true dependence of $s2$ on $s1$ caused by the reference $B(k)$. ∎

6.1.2.2 Redundant Communication Descriptors

Furthermore, communication descriptors can be eliminated if they are redundant. A communication descriptor is redundant if all elements communicated by it have already been communicated and have not been redefined.

Lemma 6.1.1:

Consider the following code fragment:

$$\begin{array}{ll}
\texttt{s1:} & \mathbf{COMM}_1(\Delta^A, <\vec{lhs}_1>, \Delta^B, <\vec{rhs}_1>, \texttt{tmp1}) \\
 & \mathbf{Mask} \rightarrow \texttt{A}(\vec{lhs}_1) = \texttt{B}(\vec{rhs}_1) \\
\texttt{s2:} & \mathbf{COMM}_2(\Delta^C, <\vec{lhs}_2>, \Delta^B, <\vec{rhs}_2>, \texttt{tmp2}) \\
 & \mathbf{Mask} \rightarrow \texttt{C}(\vec{lhs}_2) = \texttt{B}(\vec{rhs}_2)
\end{array}$$

The second communication descriptor $COMM_2$ is redundant and can be omitted if all the following conditions are satisfied:

1. $\delta^A = \delta^C$
2. $\vec{lhs}_2 \subseteq \vec{lhs}_1$
3. $\vec{rhs}_2 \subseteq \vec{rhs}_1$
4. *neither B nor $tmp1$ is redefined before statement* `s2`

∎

Proof:

The communication defined by $COMM_2$ is redundant with respect to $COMM_1$ iff all the communication sets generated by $COMM_2$ are a subset of the communication sets generated by $COMM_1$ for each processor $p \in \mathbf{P}$, i.e.

$$\forall p, q \in \mathbf{P},\ recv\_set_2(p,q) \subseteq recv\_set_1(p,q).$$

From

$$recv\_set_2(p,q) = f_2(work^C(p)) \cap local^B(q) = f_2(local^C(p) \cap \vec{lhs}_2) \cap local^B(q)$$
$$recv\_set_1(p,q) = f_1(work^A(p)) \cap local^B(q) = f_1(local^A(p) \cap \vec{lhs}_1) \cap local^B(q)$$

we obtain

$$f_2(local^C(p) \cap \vec{lhs}_2) \subseteq f_1(local^A(p) \cap \vec{lhs}_1)$$

which can be transformed to

$$f_2(local^C(p)) \cap \vec{rhs}_2 \subseteq f_1(local^A(p)) \cap \vec{rhs}_1$$

and since $\delta^A = \delta^C$ (condition (1.)) we have $\forall p \in \mathbf{P}$, $local^A(p) = local^C(p)$, and therefore

$$f_2(local^A(p)) \cap \vec{rhs}_2 \subseteq f_1(local^A(p)) \cap \vec{rhs}_1$$

which obviously is satisfied since condition (2) and (3) imply $f_2(local^A(p)) \subseteq f_1(local^A(p))$ and condition (3) states $\vec{rhs}_2 \subseteq \vec{rhs}_1$. ∎

If in the above code fragment the second communication descriptor is redundant, it can be eliminated. As a consequence, in the second assignment statement the non local elements have to be accessed from the temporary buffers generated by the first communication descriptor. This usually complicates the buffer access since the addressing mechanism now also must take the communication pattern of the first assignment statement into account. In order to correctly address the non local elements needed in the second assignment statement, it might be necessary to save the non-local index sets of the first assignment statement.

6.1.3 Fusion of Communication Descriptors

If an assignment statement contains several references to the same array, then in the basic parallelization phase a communication descriptor is generated for each reference. This however might result in communicating certain data elements more than once. In order to avoid redundant communication, different communication descriptors which refer to the same source array may be fused. This can be achieved by generating a single communication descriptor where the source access pattern is replaced by a list containing the corresponding source access patterns. This is illustrated below.

Example 6.1.2: Fusion of communication descriptors

$\textbf{COMM}\,(\Delta^A,\ < l\vec{hs} >,\ \Delta^B, < r\vec{hs}_1 >,\ \texttt{tmp\_1})$
$\textbf{COMM}\,(\Delta^A,\ < l\vec{hs} >,\ \Delta^B, < r\vec{hs}_2 >,\ \texttt{tmp\_2})$
Mask $\rightarrow$ `A(lhs) = B(1:N-1) + B(2:N)`

is transformed to

$\textbf{COMM}\,(\Delta^A,\ < l\vec{hs} >,\ \Delta^B, < r\vec{hs}_1 > \rightarrow < r\vec{hs}_2 >,\ \texttt{tmp\_2})$
Mask $\rightarrow$ `A(lhs) = B(1:N-1) + B(2:N)`

∎

Fusion of communication descriptors reduces the number of messages as well as the storage requirement for non local data at the expense of a more complicated buffer addressing mechanism.

6.1.4 Reuse of Communication Patterns

Consider the code of Example 6.1.3 that has been generated after basic parallelization and assume that A has the same distribution as C and B has the same distribution as D. If in addition the access patterns are not changed between the execution of the two assignment statements, then it is possible to reuse the send and receive sets of the first assignment statement for performing the required communication of the second assignment statement.

Example 6.1.3: Reuse of communication sets

$\textbf{COMM}\,(\Delta^A,\ < l\vec{hs} >,\ \Delta^B, < r\vec{hs} >,\ \texttt{tmp\_1})$
Mask $\rightarrow$ $\texttt{A}(l\vec{hs})\ \texttt{=}\ \texttt{B}(r\vec{hs})$
$\textbf{COMM}\,(\Delta^C,\ < l\vec{hs} >,\ \Delta^D, < r\vec{hs} >,\ \texttt{tmp\_2})$
Mask $\rightarrow$ $\texttt{C}(l\vec{hs})\ \texttt{=}\ \texttt{D}(r\vec{hs})$

∎

Whereas this optimization does not reduce the number of messages it avoids the overhead of computing send and receive sets. In Example 6.1.3 the work set of the two assignment statements is the same and therefore can be reused as well.

6.1.5 Latency Hiding

Computations involving distributed arrays can be split into two phases - one that requires only local data accesses and one that accesses non-local data which has to be received from other processors. As has been shown in Section 3.4 the work set of a particular processor with respect to an assignment statement can be split into the local work set, which does not require any communication, and the non-local work set which requires communication.

The execution of an array assignment statement can be organized in such a way that first each processor sends all the data needed by other processor and then executes the computations defined by its local work set. After that it receives non-local elements from other processors and finally performs all the computations defined by its non-local work set. This execution scheme which is also known as latency hiding [79] or latency tolerance results in overlapping of communication and computation and thus may significantly reduce the communication overhead.

Overlapping of communication and computation is achieved by splitting communication descriptors into their sending and receiving components, denoted by *COMMS* and *COMMR* (see Section 4.4.2.1), and transforming the code obtained after the basic parallelization scheme as shown in Example 6.1.4.

Example 6.1.4: Overlapping communication and computation

code after basic parallelization

COMM (Δ^A, $<\vec{lhs}>$, Δ^B, $<\vec{rhs}>$, tmp_B)
`! local references`
WHERE (Owned(A($\vec{lhs}$)) .AND. Owned(B($\vec{rhs}$))) A($\vec{lhs}$) = B($\vec{rhs}$)
`! non-local references`
WHERE (Owned(A($\vec{lhs}$)) .AND..NOT. Owned(B($\vec{rhs}$))) A($\vec{lhs}$) = tmp_B

code after splitting of communication descriptor

COMMS (Δ^A, $<\vec{lhs}>$, Δ^B, $<\vec{rhs}>$, tmp_B)
`! local references`
WHERE (Owned(A($\vec{lhs}$)) .AND. Owned(B($\vec{rhs}$))) A($\vec{lhs}$) = B($\vec{rhs}$)
COMMR (Δ^A, $<\vec{lhs}>$, Δ^B, $<\vec{rhs}>$, tmp_B)
`! non-local references`
WHERE (Owned(A($\vec{lhs}$)) .AND..NOT. Owned(B($\vec{rhs}$))) A($\vec{lhs}$) = tmp_B

∎

6.1.6 Overlap Communication

Regular computations which can be fully analyzed at compile time are processed based on the overlap concept. In the following we show how the overlap concept can be integrated with the more general buffer based approach. After the basic parallelization phase overlap analysis as described in Section 5.4 is performed.[2] Consider an assignment statement S containing references to distributed arrays on the right hand side. If for all these references an overlap description can be determined, then the corresponding communication descriptors are transformed into so-called **overlap communication descriptors** by replacing the temporary buffer by the corresponding overlap description.

The semantics of an overlap communication descriptor $COMM(\Delta^A, < \vec{lhs} >, \Delta^B, < \vec{rhs} >, OD)$ when executed on a particular processor p is shown in Figure 6.1.

```
!  send part
if local^B(p) ∩ rhs⃗ ≠ φ ∧ p = Master(local^B(p))  then
    for each q ∈ NBS(p, OD)  do
        if local^A(q) ∩ lhs⃗ ≠ φ  then
            send  B(ovp_send_q^B(p))  to q
        end if
    end for
endif

!  receive part
if local^A(p) ∩ lhs⃗ ≠ φ  then
    for each q ∈ NBR(p, OD) ∧ q = Master(local^B(q))  do
        if local^B(q) ∩ rhs⃗ ≠ φ  then
            receive  B(ovp_q^B(p))  from q
        end if
    end for
endif
```

Figure 6.1: Semantics of overlap communication descriptors

The functions $NBR(p, OD)$ and $NBS(p, OD)$ as defined in Formula 5.19 and 5.20 on page 184 determine the set of potential communication partners of p with respect to the overlap description OD.

The semantics of an overlap communication descriptors ensures that on all processors which are involved in the computation (i.e all processors that own a portion of the left hand side array section) the affected overlap areas are updated. In Example 6.1.5 the integration of the overlap concept with the more general buffer scheme is illustrated.

[2]Within this thesis we apply the overlap concept only in those cases that require nearest neighbor communication.

Example 6.1.5: Integration of overlap and buffer communication

code after basic parallelization

```
    PARAMETER(NP=8,N=4096,M=128)
    PROCESSORS R(NP)
    REAL, DISTRIBUTED(CYCLIC(M)) ::  U(N)
    REAL, ALLOCATABLE, DISTRIBUTED(CYCLIC) ::  V(:)
    ....
```
$\mathbf{COMM}_1(\Delta^U, <[2:N-1]>, \Delta^U, <[1:N-2]>, tmp1)$
$\mathbf{COMM}_2(\Delta^U, <[2:N-1]>, \Delta^U, <[2:N-1]>, tmp2)$
$\mathbf{COMM}_3(\Delta^U, <[2:N-1]>, \Delta^U, <[3:N]>, tmp3)$

s1 $Mask \rightarrow$ `U(2:N-1) = U(1:N-2) + U(2:N-1) + U(3:N)`

...

$\mathbf{COMM}_4(\Delta^V, <:>, \Delta^U, <[l:u:s]>, tmp4)$

s2 $Mask \rightarrow$ `V = U(l:u:s)`

intermediate code after generating overlap communication descriptors

....

$\mathbf{COMM}_1(\Delta^U, <[2:N-1]>, \Delta^U, <[1:N-2]>, [1/0])$
$\mathbf{COMM}_3(\Delta^U, <[2:N-1]>, \Delta^U, <[3:N]>, [0/1])$

```
s1 WHERE (OWNED(U(2:N-1))) = U(1:N-2) + U(2:N-1) + U(3:N)
   ...
```
$\mathbf{COMM}_4(\Delta^V, <:>, \Delta^U, <[l:u:s]>, tmp)$

```
s2 WHERE (OWNED(V) .AND. OWNED(U(l:u:s))) V = U(l:u:s)
s2'WHERE (OWNED(V) .AND..NOT. OWNED(U(l:u:s))) V = tmp
```

This example shows the integration of the overlap concept with the buffer based approach. Overlap analysis determines that statement s1 can be processed using overlaps. Therefore, the communication descriptors $COMM_1$ and $COMM_3$ are transformed into overlap communication descriptors. $COMM_2$ can be removed since it does not generate communication. As can be seen it is not necessary to split s2 into the local and non-local parts. For statement s2 however the overlap concept cannot be applied since the subscript triplet [l:u:s] cannot be analyzed by the compiler and, as a consequence, overlap analysis fails. Therefore, s2 is processed by applying the buffer based approach, and the assignment is split into local and non-local parts. ∎

6.1.7 Storage Optimization

In order to reduce the storage requirement of the target program, it is fundamental to allocate on each processor only memory for that part of an array that is owned by the processor. Furthermore, storage for non-local data must be provided. As already mentioned, allocating only the local segment of a distributed arrays usually implies that global indices have to be translated into corresponding local indices. If this index conversion is carried out on a single element basis, a large overhead may be introduced. Therefore, it is crucial to apply the index conversion to whole array sections wherever possible. In Section 4.2.2 it has been shown how (global) sections of a distributed array can be translated into sections of the corresponding local segment on a particular processor. If enough compile time information is available, local storage for distributed arrays may be statically allocated. If this is not the case, storage allocation has to be deferred until runtime and arrays must be dynamically allocated.

In order to choose the most appropriate scheme for handling accesses to non-local data, analysis information about all references to a distributed arrays and about the actual distribution is necessary. Within this thesis we concentrate on two different storage schemes for non-local data: overlaps and buffers. Overlaps are especially useful for computations that exhibit a large degree of locality, as it is the case for shift or stencil operations. Temporary buffers are needed if the amount of non-local data accessed with regard to a particular array is unknown at the time the array is allocated, or if the non-local elements cannot be restricted to the boundary of the local segments. Another method for handling non-local data, which is used in the case of sparse or irregular computations are hash tables. This approach is taken in the Parti runtime primitives [41].

6.2 Structure of Target Code

6.2.1 Declaration Statements

In the following we assume that the number of processors on which the program should execute is not known until run-time. All distributed arrays of the input program have to be analyzed in order to determine their resulting layout in the final SPMD program. This analysis requires to compute the local index set on a particular processor according to the formulas of Section 4.2.1.

However, the size of the local index set usually cannot be determined at compile time since it depends on the total number of processors P on which the program will execute. As a consequence, all statically allocated arrays of the input program have to be transformed into dynamically allocated (*ALLOCATABLE*) arrays in the SPMD program. Furthermore, code has to be inserted before the first executable statement of the source program in order to determine on each processor the local segments of distributed arrays, to perform dynamic memory allocation, and to set up layout descriptors. We refer to this phase as *data layout phase*. For *ALLOCATABLE* arrays of the input program the data layout phase has to be performed each time an allocation instance is initiated.

We first describe the general approach used for the transformation of array declarations and then discuss special cases.

6.2.1.1 Dynamic Array Allocation

Every declaration of a distributed array of the input program is transformed into a declaration of an *ALLOCATABLE* array. Prior to the first executable statement code for the dynamic memory allocation is inserted. On each processor the array is allocated using local indices with lower bound 1 and upper bound equal to the size of the corresponding local index set (see Section 4.2.1). Furthermore, every reference to a distributed array, has to be transformed in order to convert global indices into local ones using the index conversion functions defined in Section 4.2.2. Note that as a consequence of this scheme the size of the local segment of the "same" array on different processors may vary.

Example 6.2.1: Modification of array declarations

original declaration ...

```
PROCESSORS R(NUMBER_OF_PROCESSORS)
REAL, DISTRIBUTED(CYCLIC(M)) TO R ::  A(N)
```

is transformed to ...

```
REAL, ALLOCATABLE ::  A(:)
!  compute size of local index set
LOC_SIZE = VF_Local_Size(A_ld,MY_PROC)
...
!  layout phase for A
ALLOCATE(A(SIZE(LOC_SIZE)))
```

■

6.2.1.2 Static Array Allocation

In the case where the number of processors is a compile-time constant, a statically allocated distributed array can be transformed in such a way that in the SPMD program its size equals the maximum size of the local index set of all processors. This avoids to transform every array of the input program into an ALLOCATABLE array. This method can also be applied in the case of explicit shape dummy arrays with inherited distribution by passing the actual bounds of the local segment as arguments. However, this approach is only applicable if the rank of actual and formal array are the same.

Array Allocation Using Global Indices

Arrays with a $BLOCK(M)$ distribution can be allocated in the resulting SPMD program using global indices. This has the advantage that no global to local index conversion as described in Section 4.2.2 has to be applied.

6.2.2 Array Assignment Statements

The final implementation of array assignments has to deal with the following issues: computation of work and communication sets, index conversion, coordinating interprocessor communication, managing temporary buffers and accesses to copies of non-local data.

In the following we assume that access to non-local data is handled by means of dynamically allocated temporary buffers. In Figure 6.2 a Fortran 90 SPMD pseudo-code on a particular processor p, $1 \leq p \leq n\_proc$, for a multidimensional array assignment statement of the form

$$A(l_{l_1} : u_{l_1} : s_{l_1}, \ldots, l_{l_n} : u_{l_n} : s_{l_n}) = B(l_{r_1} : u_{r_1} : s_{r_1}, \ldots, l_{r_n} : u_{r_n} : s_{r_n})$$

is illustrated. It is assumed that both arrays A and B have a block or a cyclic distribution. In this case, communication sets can always be represented by means of a single array section. Furthermore, we assume for sake of illustration that both arrays have the same dimensionality.

The set of elements (i.e. $send^B(p, q)$) that have to be sent by a processor p to a processor q are computed by means of the procedure VF_Send_Set[3]. Similarly, the procedure VF_Recv_Set determines the receive index sets. In Figure indices of those elements of A that have to be updated with non-local values of B are denoted by the triplet `[l1:u1:s1,...,ln:un:sn]`, whereas the corresponding elements of B are denoted by `[lr1:ur1:sr1,...,lrn:urn:srn]`. Input arguments to these functions are the processor number of the communication partner and layout descriptors of the *lhs* and *rhs* array, respectively. Both functions return the communication sets with respect to local indices, represented as triplets by means of variables of derived type.

For packing messages into a contiguous message buffer the intrinsic function *RESHAPE* is utilized. The procedure VF_Local_Work, returns those indices for which the assignment can be executed without communication. Note that the local work set on a processor p is given by $send(p, p)$. Addressing of the non-local elements in the buffer is achieved by means of the Fortran 90 intrinsic functions *RESHAPE* and *SHAPE*.

In the case of block cyclic distributions, communication sets are generally specified by more than one section. In this case, it is necessary to pack sub-messages described by a single section in one buffer in order to reduce the number of messages. Furthermore, there must be an additional loop for the send part, the receive part and the computation parts.

In Figure 6.2 messages from different processors to a particular processor p are managed by separate buffers. Since all these buffers have to be dynamically allocated, this may introduce a large overhead.

If only one message buffer should be used, management of message buffers becomes more complicated. The size of such a message buffer on a particular processor p is given by $\sum_{q=1}^{P} Size(recv(p, q))$. The position where elements according to $recv^B(p, q)$ for a particular processor q is stored can be managed on each processor via a temporary index array of size equal to the number of processors.

[3]We assume here that the function returns `send_size = 0` if $q = p$. The same holds for VF_Recv_Set

```
!compute send sets; send data
DO q=1, n_proc
   CALL VF_Send_Set (q, A_ld, B_ld, send_set, send_size)
   IF (send_size > 0) THEN
      ALLOCATE (tmp_B(send_size))

      l1 = send_set%l(1); u1 = send_set%u(1); s1 = send_set%s(1)
            ...........
      ln = send_set%l(n); un = send_set%u(n); sn = send_set%s(n)

      !store data contiguously in buffer and send buffer
      tmp_B = RESHAPE (B(l1:u1:s1, ..., ln:un:sn),(/ send_size /))
      send tmp_B to q

      DEALLOCATE (tmp_B)
   END IF
END DO

!compute local work set
CALL VF_Local_Work(loc_wrk, A_ld, B_ld)

l1 = loc_wrk%l(1); u1 = loc_wrk%u(1); s1 = loc_wrk%s(1)
lr1 = loc_wrk%lr(1); u1 = loc_wrk%ur(1); sr1 = loc_wrk%s(1)
         ...........
ln = loc_wrk%l(n); un = loc_wrk%u(n); sn = loc_wrk%s(n)
lrn = loc_wrk%l(n); urn = loc_wrk%u(n); srn = loc_wrk%s(n)

!execute local assignments
A(l1:u1:s1, ..., ln:un:sn) = B(lr1:ur1:sr1,...,lrn:urn:srn)

!compute receive sets; receive data;
DO q=1, n_proc
   CALL VF_Recv_Set(q,A_ld,B_ld,recv_set,recv_size)
   IF (recv_size > 0) THEN
      ALLOCATE (tmp_B(recv_size))
      receive tmp_B from q

      l1 = recv_set%l(1); u1 = recv_set%u(1); s1 = recv_set%s(1)
      lr1 = recv_set%l(1); ur1 = recv_set%u(1); sr1 = recv_set%s(1)
            ...........
      ln = recv_set%l(n); un = recv_set%u(n); sn = recv_set%s(n)
      lrn = recv_set%l(n); urn = recv_set%u(n); srn = recv_set%s(n)

      !execute non-local assignments from buffer
      A(l1:u1:s1,...,ln:un:sn) = RESHAPE(tmp_B,(/SHAPE(B(lr1:ur1:sr1,...,lrn:urn:srn))/))
      DEALLOCATE (tmp_B)
   END IF
END DO
```

Figure 6.2: Scheme for the implementation of array assignments

Chapter 7

Related Work

Research in the field of compiling high level languages for distributed memory machines started in the second half of the 80's [136, 137, 27]. Since then, work in this area has grown rapidly and a number of prototype compilation systems have been developed. In this section we examine alternative approaches taken by other researches with respect to both language development and compiler support for distributed memory machines. In the first part of this chapter we discuss related approaches with respect to languages and language extensions that aim to alleviate programming of distributed memory machines. In the second part we focus on compilers, transformation systems, and compilation techniques for distributed memory machines.

7.1 Language Support

Blaze, Kali

Kali [93] and its predecessor Blaze [92] are extensions to Fortran-like sequential languages, that provide processor array declarations, data distribution and a *forall* loop for specifying parallel computations within a global address space. Kali was the first language that introduced data distribution annotations and had a significant influence on Vienna Fortran. Besides block, cyclic and block-cyclic distribution, it provides support for irregular distributions and simple forms of user-defined distributions. However, Kali is not completely specified, in particular with respect to procedure interfaces, and does not support alignment and dynamic data distribution. Compilation techniques developed within the Kali project are discussed in the next section.

Booster

Booster, developed by Paalvast et.al. [100, 101], is a data parallel language offering a global address space and conventional imperative form of control flow. This new and experimental language is based on a set-based calculus, called *view-calculus*. Multidimensional objects have to be declared by means of a *shape* declaration, and may be manipulated similar to Fortran 90 arrays. Moreover, Booster provides a special construct, called *view*, together with a number of special operators for the explicit manipulation of the index domains of multidimensional objects. The view concept can be understood as a generalization of the Fortran 8x [96] identify statement. Booster offers two selection functions for the computation of minimum and maximum of shapes and a reduction function. Furthermore, Booster provides a separate annotation language for defining a virtual machine, and for specifying the mapping of a virtual machine to a real machine. Data as well as computation distribution may be specified by means of *view functions*. Booster, however, does not support dynamic memory allocation and a general mechanism for user defined data structures. Research is currently concentrated on compiling Booster into Fortran SPMD code within a rule-based compilation system [20, 21].

CM Fortran, C*

CM Fortran [117, 118], initially developed for the Connection Machine CM-2 and recently ported to the CM-5, is a based on standard Fortran 77 with extensions for data parallel programming. Besides the Fortran 90 array-processing extensions, layout directives, a *FORALL* statement, and additional intrinsic functions in order to support data-parallel programming are provided. However, the features for dynamic memory allocation of Fortran 90 are not provided. Arrays for which no layout directives are specified and which are referenced in Fortran 77 style are stored together with scalar data on the partition manager of the CM-5. Arrays for which explicit layout directives are provided or which are referenced as whole objects in Fortran 90 style are distributed over the processing nodes.

C* [118] is a data-parallel extension of ANSI-C for the Connection Machine. The user can declare parallel variables by providing a *shape* declaration. These variables are then distributed over the processing nodes of the machine. C* has a number of new statements for operations on parallel data. The *WITH* statement is provided to specify data parallel operations on variables of a particular shape similar to Fortran 90 array operations. Furthermore, a *WHERE* statement and new operators for computing the minimum or maximum of parallel variables are provided. The *pcoord* function and the concept of *left indexing* is provided for specifying communication along the axis of a shape.

CRAFT

The Cray Fortran programming model (CRAFT) is a set of language extensions to Cray Fortran (cf77) [40, 103] targeting the Cray T3D massively parallel processor. The CRAFT programming model enables the user to specify both the distribution of data and work. Data objects are either private to a task or shared among all tasks. It is assumed that only one task executes on each processing element. Private objects are allocated on each PE but are not kept consistent. Dimensional distributions are specified by supplying a weight to each array dimension which determines the number of processors over which the dimension will be distributed. Furthermore, canonical distributions, which can be thought of as block-cyclic distributions of a linearized array, are provided. In order to reduce the overhead of address calculation, the size of each array dimension, the block sizes and the number of processors is transformed to a power of two.

Commonly used distributions may be associated with a name by declaring a *geometry* similar to Vienna Fortran distribution type definitions. Common blocks may be distributed either as a whole or a distribution may be specified for individual objects in a common block. In the latter case, however, the storage association implied by common blocks is not retained. Dynamic data redistribution is not possible, but distributions may be changed at procedure boundaries.

Shared loops that execute in parallel and methods for distributing loop iterations, either pre-scheduling or self-scheduling, are provided. The iterations of a pre-scheduled loop may be distributed either uniformly or by means of alignment using an *ON* clause. For self-scheduled shared loops two block scheduling mechanisms and a guided self-scheduling mechanism are provided. Many features of shared memory parallel languages have been retained: these include critical sections, barriers, events and locks. Intrinsic functions provide access to parts of arrays local to a processor, and reduction and parallel prefix operations are included. Furthermore, parallel I/O is supported.

Yale Extensions

In the Yale Extensions, a layout specification language for Fortran 90 developed by Chen et al. [37], the distribution of arrays is specified in three stages: alignment, partition and physical map. Since all these three stages are modeled as domain morphisms, which have been taken over from the functional language CRYSTAL [88], data replication cannot be specified. Alignment is expressed by means of Fortran 90 array intrinsic functions. The scope of layout directives is restricted to phases and thus a block structure is imposed on Fortran 90. Map operators for embedding the logical machine into the target machine are provided.

Fortran D, Fortran 90D

The programming language Fortran D [48], developed at Rice University and Syracuse University, proposes a Fortran language extension in which the programmer specifies the distribution of data by aligning each array to a virtual array, known as a decomposition, and then specifying a distribution of the decomposition. Block, cyclic, block-cyclic, and irregular distributions using map arrays may be specified. Processor arrays are not accessible to the programmer but are created implicitly. As a consequence, distribution with respect to processor subsets is not possible.

Since the layout of data only may be specified by means of executable statements, array distributions are dynamic only. Fortran D does not provide any support for the user to control the mechanisms of dynamic data distribution and does not define the semanitcs of transferring distributed arguments at procedure boundaries.

Fortran 90D [133] is a subset of Fortran 90 (including the array features of Fortran 90) with Fortran D extensions.

High Performance Fortran

High Performance Fortran (HPF) was developed by the *High Performance Fortran Forum*, a consortium from industry, government labs and academia, in order to provide a standard set of extensions to support programming on a wide variety of parallel architectures, including massively parallel SIMD and MIMD systems and vector processors. The main concepts in HPF have been derived from a number of predecessor languages, including mainly Kali [93], CM Fortran [117], Fortran D [48], and Vienna Fortran [139]. HPF is designed as a set of extensions (mainly in the form of directives) to Fortran 90.

The HPF language model includes, besides the basic features for data distribution and alignment, the concept of *templates*. Templates, similar to Fortran D decompositions, are basically a named index domain that can be used for specifying alignment between arrays. The data distribution model of HPF is based on a three-level mapping. Arrays are mapped to templates which in turn are mapped to virtual processor grids. Virtual processor grids in turn are mapped to the physical processors. This last mapping, however, is not specified within HPF. HPF assumes that each array is aligned to a template, either explicitly or implicitly. However, this causes problems in the context of allocatable arrays since templates cannot be allocatable. Furthermore, templates significantly complicate the semantics of argument transfer at procedure boundaries.

HPF has adopted the concept of dynamically and statically distributed arrays from Vienna Fortran. However, in HPF it is not possible to change the distribution of an array as the result of a procedure call. The features for data distribution are similar to the basic distribution features of Vienna Fortran. More advanced concepts like distribution to processor array sections, indirect distributions and user-defined distributions are not included in HPF.

With exception of the *FORALL* statement all HPF extensions are specified in the form of compiler directives. The semantics of the *FORALL* statement is the same as in CM-Fortran (SIMD-semantics). In order to allow calls to procedures from inside of a *FORALL*, HPF provides *pure procedures* which do not produce side effects. The user may assert that the iterations of a loop can be executed independently by means of the *INDEPENDENT* attribute. Explicit interfaces to procedures written in other programming paradigms may be defined using the concept of *extrinsic procedures*.

Pointers may be the object of a redistribute or realign statement only if they are associated with a whole array, not an array section. The semantics of pointer assignments in the context of distributed arrays is currently undefined. Furthermore, HPF does not provide any support for the distribution of components of derived types. A detailed comparison of Vienna Fortran 90 and HPF can be found in [13].

7.2 Parallelizing Compilers

SUPERB, VFCS

SUPERB [136, 137] is a semi-automatic parallelizer for standard Fortran 77 programs, initially developed at the University of Bonn. SUPERB was the first implemented restructuring tool of this kind and supports both automatic parallelization as well as vectorization. The system takes, in addition to a sequential Fortran program, a specification of the desired data distribution and converts the code to an equivalent program to run on a distributed memory machine, inserting the communication required and optimizing it where possible. The user is able to specify arbitrary multidimensional block distributions. Besides data dependence analysis and interprocedural data flow analysis the system performs interprocedural distribution and overlap analysis, interprocedural distribution propagation, procedure cloning. Several optimizations such as message vectorization, elimination of redundant communication and communication fusion have been integrated.

The Vienna Fortran Compilation System (VFCS) [34], which is a further development of SUPERB currently handles much of the functionality of Vienna Fortran. The functionality of the system significantly extends beyond present HPF. VFCS currently handles both Vienna Fortran and subset HPF programs. Furthermore, a subset of Vienna Fortran 90 has been implemented. Static and dynamic distributions are provided. Loops with regular data accesses that can be analyzed at compile are parallelized very efficiently based on the overlap concept. For irregular data accesses, run-time compilation techniques based on the *inspector-executor* paradigm [41] have been integrated. Besides a facility for user-defined distributions, the VFCS is able to efficiently handle Fortran workspaces [52].

The VFCS can be used as a command line compiler for Vienna Fortran or HPF programs, or as an interactive parallelization system where the parallelization process is guided by user decisions. Furthermore, the VFCS can be used for analyzing and optimizing sequential Fortran programs. A number of program transformations are provided. Among these are induction variable substitution, interprocedural constant propagation, inline expansion, loop distribution, loop interchange, and others.

The information services of the VFCS compute analysis information and provide access to the program information set. The dependence analysis service computes for every definition of a variable both the true dependences and the output dependences, and for every use of a variable the anti dependences. The user can look at all the dependences between all the statements in the body of a DO-loop. The data flow analysis service computes use/def- and def/use- chains, anti chains and output chains. The call graph can be displayed.

Moreover, the VFCS offers an integrated weight-finder [43], i.e. a sequential program profiler, to determine the performance-relevant code regions and provides support for predicting the performance of parallel programs [42, 44] and for performance measurement [129].

The VFCS currently generates message passing code for the iPSC/860, the Intel Paragon, and PARMACS [18], a programming interface for parallel computers. Code generation for the Meiko CS-2 is currently under development.

Pandore

The Pandore system [4] takes as input a sequential C program, annotated with a virtual machine declaration and data distribution specifications, and generates parallel C code augmented with calls to message passing primitives. In addition to block and cyclic distributions, so-called block-overlap distributions may be specified. Block-overlap distributions have the effect that boundary elements are allocated on more than one processor. However, the overlaps are not determined automatically by the system but have to be explicitly specified by the user. The virtual machine may be configured as a vector, ring, grid or torus.

The compilation process is based on the SPMD paradigm and the owner computes rule is applied exclusively. In order to deal with remote data accesses, each assignment statement containing references to distributed variables is translated into two intermediate instructions, *Refresh* and *Exec*. The *Refresh* instruction is responsible for updating all variables read by the assignment statement. The use of the *Exec* statement it is enforces that each processor only executes the assignment for variables that are owned by it. The current prototype implementation only supports single element messages i.e there is no support for message vectorization. In addition, during execution of the *Exec* statement each processor has to determine at runtime if it owns the corresponding variable.

As a consequence, the generated parallel program may be very inefficient [5]. Optimization with respect to communication vectorization and work distribution are currently under development.

Kali Compiler

The Kali compiler [79, 82, 78, 93] was the first system that supported both regular and irregular computations. Kali requires the user to declare a virtual processor array and to specify both the distribution of arrays and the distribution of loop iterations. Besides *BLOCK* and *CYCLIC* distributions Kali has a mechanism for specifying user defined distributions. Moreover, distributions may be specified with respect to processor subsets and programs may be parameterized by the number of processors. The distribution of formal procedure arguments must be specified explicitly.

Parallelism is expressed by means of the *FORALL* statement. The user is required to explicitly specify the distribution of the *FORALL* iterations by means of an *ON*-clause. The necessary communication is automatically generated by the compiler. Since *FORALL*s have no loop carried dependencies, iterations can be executed concurrently and communication can be vectorized. Non-local data is managed by means of temporary buffers. The parallel execution scheme of FORALL statements is designed to minimize the communication overhead by overlapping communication with local computations.

The compilation process is based on the SPMD model. From the data distribution specification the compiler generates functions to determine the set of elements owned by each processor. The distribution of forall iterations is represented by means of execution sets. Based on the set of local elements and the execution set, communication sets are determined. In the absence of enough compile time information these sets have to be computed at runtime. By examining simple array access patterns Koelbel [81] has worked out a lot of special cases which can be handled efficiently. In cases where compile time analysis fails (i.e indirect array accesses), run-time compilation techniques based on the inspector/executor strategy are applied [80].

Crystal Compiler

Crystal is a high-level functional language that contains special constructs for expressing data parallelism, i.e. index domains, forall loops, and APL-like aggregate operators [85, 86, 87, 88]. The Crystal compiler system supports both automatic data distribution as well as collective communication generation and performs significant compile-time analysis and optimizations.

In the first phase of the compilation process the program is decomposed into π-blocks. For each π-block the so-called *component affinity graph* is constructed. The component affinity graph serves as the basis for index domain alignment [85], which aims to minimize the cost of cross-references between distributed arrays. Heuristics are employed to obtain alignment information and the source program is transformed into an aligned Crystal program, considering simple alignment functions like *shift*, *embedding*, *permutation*, and *reflection*.

In the second phase data partitioning and communication generation is performed. The two data partition strategies considered, block and interleaved block partitions, decompose an array index domain into subdomains of equal size and shape which are mapped onto an abstract target machine. Initially the compiler generates a parameterized data layout strategy. The subsequent process of communication generation is based on syntactic pattern matching. For each assignment a communication pattern is determined and matched with a set of communication routines. The matching algorithm is guided by a cost metric which describes the execution time of the communication routines, i.e. uniform shift, one-all-broadcast, all-one-reduce, point-to-point, and affine transform. Reference patterns which cannot be perfectly matched with a single communication routine are composed into subpatterns, and the matching algorithm is applied recursively to these subpatterns. After the communication routines have been selected the final data layout is determined based on the estimated cost of computation and communication.

Finally a message passing C program for the Intel IPSC/2, NCube, or Warp systolic machine is generated.

The Fortran D Compiler

The Fortran D compiler [68, 120] converts restricted Fortran D programs into SPMD node programs with explicit message passing calls. Parallelization is based on the owner computes rule. The basic compilation techniques are similar to those developed within the SUPERB project. The Fortran D compiler is integrated with the ParaScope programming environment which supplies information from data flow and dependence analysis information and provides support for loop transformations such as *loop distribution, loop interchange*, and *strip mining*. Furthermore, support for performance estimation based on so-called training sets is provided. Several communication optimization techniques like *communication vectorization, vector message pipelining, message coalescing, message aggregation* and exploitation of collective communication have been investigated [66, 67]. Storage for non-local data is handled via buffers or overlaps. Run-time compilation techniques [41] are applied for irregular computations.

The current prototype Fortran D compiler assumes that loop bounds, subscript functions and data distributions are always known at compile time [68]. Furthermore, distributions are restricted to one array dimension and simple alignment functions (i.e only alignment offsets), and the number of processors has to be fixed at compile time.

ADAPT

The ADAPT system [94, 95], under development at the University of Southampton, transforms *Distributed Fortran 90* (DF90) into standard Fortran 77 programs with embedded calls to the ADLIB library, which provides communication routines and customized Fortran 90 intrinsic procedures. DF90 is a subset of Fortran 90, with extensions for data distribution. Data distribution is restricted to block distribution. The distribution of formal procedure arguments may be specified explicitly or assumed from the actual argument. Alignment, dynamic memory allocation and dynamic data redistribution is not supported. A process grid declaration has to be supplied in a separate file. It is required that the shape of the process grid is known at compile-time.

The parallelization process is based on the SPMD programming model, and aims to exploit the parallelism inherent in array assignments. A different approach compared to ours is taken for the parallelization of array assignments. First the compiler checks if the array section on the left hand side is co-distributed with the right hand side array section. If this is not the case, the right hand side section is redistributed into a temporary array, and the assignment statement is transformed accordingly, which may lead to a large memory overhead. Procedures for redistribution of array sections as well as routines corresponding to Fortran 90 intrinsic functions are contained in the ADLIB library. The ADLIB library contains routines that perform redistribution of regular and irregular array sections, and procedures which implement a subset of the Fortran 90 array intrinsic functions that are called with distributed data. Each array argument passed to a library procedure is accompanied by an *attribute vector* which contains the essential information concerning its distribution. The ADLIB library is currently under development and targets the T9000+C104 message passing architecture and the PARMACS macros [18].

Oxygen

Oxygen [108] is an automatic parallelization system for the K2 distributed memory parallel machine which has a torus topology. The design philosophy of Oxygen is based on the fact that communication on the K2 is relatively inexpensive. The parallelization strategy is characterized by a decomposition of the program structure as well as a decomposition of the data structure which is achieved by inserting special directives. Initially the sequential Fortran program is decomposed into a sequence of code blocks which are classified either as local or as public. Whereas local blocks can be executed without the need of interprocessor communication, public blocks (*P_BLOCK*) require access to non-local data. Interprocessor communication is automatically generated by inserting code, so-called symbol handlers, in front of each *P_BLOCK*. Loops are parallelized by distribution the iteration space with the *split* or *scatter* directive and mapping the iteration sub-spaces with the directives *ring, rowwise*, or *colwise* to the processing elements. Distributed arrays have to be classified either as *local*, *multicopy* or *singlecopy*. Ownership of *multicopy* data changes dynamically.

Explicit communication routines are provided for broadcast and reductions. In order to parallelize loops with indirect references, runtime consistency checks are performed. Currently the Oxygen system transforms Fortran programs with user specified directives into C++ message passing programs. Implementation of a preprocessor that generates the compiler directives automatically is underway.

MIMDizer, Forge90

The Forge90 system [6], a further development of the MIMDizer [65, 98], from Applied Parallel Research, is an interactive parallelization system for Fortran targeting shared and distributed memory MIMD machines. Forge90 provides tools that perform data dependence analysis, data flow analysis and interprocedural analysis. In addition, it provides support for code instrumentation and profiling. The input language is subset-HPF with additional restrictions, i.e. only a single array dimension may be distributed, block-cyclic distributions and alignment containing strides are not supported. Moreover, variable sized arrays cannot be distributed. Initially Fortran 90 constructs are transformed by a preprocessor into Fortran 77. Loops that are candidates for parallelization may be interactively marked by the user. Communication for non-local memory accesses is automatically handled by the system by inserting appropriate calls to the FORGE message passing library.

The output SPMD program has to be compiled with the target machine compiler and linked with the run time library. In order to simplify ownership calculations at run time, distributed arrays are linearized. Currently Forge90 provides communication libraries for native INTEL, native nCube, PVM, EXPRESS and P4.

Adaptor

The Adaptor system [19] developed at the GMD is a prototype system for interactive source-to-source transformation for Fortran programs. The input language is Fortran 77 with Fortran 90 array extensions, and data distribution directives. The output is a Fortran 77 SPMD program with explicit message passing calls. Only block and cyclic distributions with respect to a single array dimension are supported. Alignment and inheritance of distributions at procedure boundaries is not supported. The system has no facilities for automatic parallelization (i.e. no dependence analysis) and only exploits the parallelism inherent in array assignment statements.

Fortran90D Compiler

The Fortran90D compiler [133] is a source-to-source compiler for SIMD and MIMD parallel machines. The Fortran90D compiler only exploits the parallelism inherent in the array constructs and makes no attempts to parallelize sequential constructs (i.e loops). Array assignments are transformed by the front end into FORALL loops. Data distribution is restricted to static block and cyclic distributions of a single array dimension. Allocatable arrays, automatic arrays and array valued functions are not yet supported. The number of processors on which the program executes has to be specified at compile time.

The output is a Fortran 77 SPMD program with calls to a communication library. The system provides support for collective communication and performs optimizations such as overlapping communication and computation and communication vectorization. Runtime compilation techniques based on the PARTI primitives [41] are applied in case of indirect array references.

7.3 Compilation Techniques

Stichnoth [115] describes techniques for the parallelization of HPF array assignment statements. His work however does not treat the full functionality provided by the HPF distribution directives. In particular alignment functions are restricted to shifts only, i.e. the stride of the alignment function must be 1.

The main part of this work deals with block-cyclic distributions. Optimizations for block and cyclic distributions are discussed. The set of local elements of an array with a block cyclic distribution is represented as a union of non-contiguous sections similar to the cyclic-representation of page 95. Algorithms that determine the send and receive sets either in global or local indices, and a function for converting global to local indices are provided. Basically these algorithms are based on the intersection of regular sections.

Chatterjee et.al [29] present a general solution for generating local addresses for arrays that have a block-cyclic distribution. Given an array A with $CYCLIC(M)$ distribution the sequence of local memory accesses for an array section $A(l:u:s)$ can be characterized by a finite state machine with at most M states, where the states represent the offsets between consecutively accessed elements of the array section in the local memory of a processor. They provide algorithms to compute the essential information about the finite state machine and extend their method by considering arbitrary affine alignment functions. The preprocessing overhead for generating the finite state machine includes finding the first element of the array section on a particular processor, which is achieved by solving a set of linear diophantine equations. After sorting all the solutions the local memory access sequence is constructed.

However, they do not derive a general method for converting global indices into local indices. As a consequence, the preprocessing overhead for generating the finite state machine has to be paid even if a single array index has to be converted. Unlike in our approach (see Section 4.2.4), the generation of communication sets is done on a single element basis by scanning over the data in the local memory of a processor, and determining for each single element whether it is needed by another processor.

Gupta et.al [58] develop techniques for the compilation of array assignment statements for execution on distributed memory machines. They present closed form expressions to represent the local memory access sequence and communication sets for block or cylic distributions. They extend these methods to block-cyclic distributions based on a virtual processor approach. By introducing virtual processors a block-cyclic distribution is treated either as a block or as a cyclic distribution with respect to virtual processors. Whereas their techniques are similar to those presented in this thesis they do not handle alignment.

Paalvast et.al. [99] present techniques to determine the active elements of a regular array section, i.e. those elements that are owned by a particular processor, with respect to block, cyclic and block-cyclic distributions. Communication sets are generated on a single element basis by scanning over the active elements and determining for each element if it has to be communicated. This scheme may result in a high run-time overhead. Later, Sips [113] extends these techniques to arrays that are aligned with a distributed template. In this work linear as well as piecewise monotonic alignment functions can be handled, and techniques for index conversion and memory management, similar to our work, are presented. However, no solutions for communication set generation are provided.

Chapter 8

Conclusions

In this thesis we have presented Vienna Fortran 90, a highly advanced data parallel language for machine independent programming of distributed memory parallel computers. Furthermore, we have developed a general framework and new techniques for compiling Vienna Fortran 90 for distributed memory parallel computers.

8.1 Contributions

This work has made the following contributions in the area of language design and compilation for distributed memory computers.

Language Design

We have extended and generalized the language features of Vienna Fortran with respect to Fortran 90 and have developed a number of new concepts that significantly improve the expressiveness and applicability of this language. The contributions of this thesis with respect to language design are the following:

- *Generalization of Vienna Fortran to Fortran 90*
  - Whereas many language features of Vienna Fortran 90 have been taken over from Vienna Fortran a number of language constructs and mechanisms have been redesigned, extended or clarified.
  - The concept of indirect distributions has been generalized such that it is possible to distribute either single array dimensions or whole arrays indirectly.
  - Distribution with respect to processor subsets can be specified both with respect to regular processor array sections as well as irregular (i.e vector subscripted) processor array sections.
  - The Fortran 90 intrinsic function *RESHAPE* may be used with a processor array as argument in order to provide different views of the available set of processors.
  - A new form of alignment specification based on array sections and the *ALIGN* statement has been introduced. The impact of dynamically allocated arrays in conjunction with alignment has been clarified.

- *Generalization of user defined distribution and alignment functions*

  The mechanisms provided by Vienna Fortran for the specification of arbitrary user defined alignment and distribution functions has been unified and generalized by introducing the concept of *mapping procedures*. Mapping procedures are similar to Fortran 90 procedures, may have arguments, may be recursive and may be passed as arguments to mapping procedures.

  Besides the specification of arbitrary user defined distribution and alignment functions they may be used to define arbitrary mappings of *FORALL* loop iterations to processor sections.

- *Distribution of derived types*

  The concept of distributed derived types makes it possible to specify derived types with either statically or dynamically distributed components.

  Furthermore, we have developed a concept in order to statically distribute objects of the same distributed derived type to different processor subsets.

- *Distribution of pointers*

  We have extended the distribution concepts to pointer objects and have defined the semantics of pointer assignments with respect to distributed variables.

  The impact of distributed pointer variables in the context of procedure interfaces has been analyzed.

- *Distribution type definitions*

  The concept of distribution type definitions has been redesigned and generalized.

- *Local procedures*

  Local procedures alleviating the reuse of sequential kernels and providing an interface to other sequential languages have been specified.

Compilation Techniques

This thesis has developed a general framework for the compilation of Vienna Fortran 90 programs. We have developed new techniques and extended and integrated existing methods. Much of this work is applicable to other languages, in particular to HPF.

- *New parallelization scheme*

  We have developed a new parallelization scheme, based on high level communication descriptors, and described the basic parallelization of executable statements including procedure calls.

  Communication descriptors establish an abstraction of the actual communication and thus provide a suitable means for many optimization techniques. They allow a uniform treatment of references to scalars and arrays.

- *Runtime support*

  We have developed mathematical functions, parameterized by the number of processors, the array size, and the actual distribution, that determine the local index set of distributed arrays, the work distribution and the communication sets for assignment statements. Furthermore, we have developed functions in order to convert global to local indices and vice versa. These functions are either used by the compiler or, in the absence of enough compile time information, they are included in the run time system. All these functions have been validated by means of prototype implementation.

- *Compilation of array assignments*

  We have developed new concepts for the compilation of array assignment statements that aim in exploiting both inter-processor parallelism as well as intra-processor parallelism by maintaining the structure of array assignment statements as far as possible.

- *Management of non-local data*

  Management of non-local data is achieved either by allocating temporary buffers or by extending the local segment of a distributed array by overlaps.

  These two concepts have been integrated such that both schemes can be used for different references to the same array.

- *New techniques for block-cyclic distributions*

  Different techniques for the management of block-cyclic distributions have been examined. New methods for implicit (i.e aligned) block-cyclic distributions with respect to memory allocation, index conversion, communication set generation and code generation have been developed.

  The overlap concept has been extended to block-cyclic distributions, and overlap communication descriptors have been defined.

- *Optimization techniques*

  Several basic optimization techniques have been examined. In particular, we have shown that the concept of communication descriptors provides a suitable means for performing optimization of communication.

- *Implementation of selected language features*

  A number of selected language elements has been integrated in the Vienna Fortran Compilation Systems. This includes multidimensional block and generalized block distributions, distribution by alignment allowing arbitrary linear alignment functions, distributions with respect to processor subsets and array assignment statements.

  The above mentioned Vienna Fortran 90 runtime support functions, including runtime representation of distributions, have been implemented in a prototype runtime support system written in Fortran 90.

8.2 Future Research

Automatic Data Distribution

Selecting a data distribution is the most crucial step in the parallelization of data placement languages like Vienna Fortran 90, and strongly influences the overall performance of the parallelized program. Therefore, the provision of automatic support for data distribution is an important future research topic. Currently only partial solutions to this problem exist. In particular, Li and Chen [85] have developed methods for automatically aligning arrays in the context of the Crystal project; Ramanujam and Sadayappan [104] propose analytical methods for determining the data distribution of arrays with respect do single do-loops by deriving a system of linear equations based on the access patterns. Gupta and Banerjee developed [56, 57] methods for deriving data distributions based on pattern matching techniques and communication metrics. Currently support for automatic data distribution based on analytical methods, performance prediction and heuristics is being implemented within the Vienna Fortran compiler [33].

Support for Parallel I/O

One important area of future research is the provision of language support for parallel input/output operations. In particular, we will extend the parallel I/O features provided by Vienna Fortran to Vienna Fortran 90. Currently Vienna Fortran provides language support to read and write distributed data from/to a concurrent file system (e.g. the CFS on the Intel iPSC/860) by adding a distribution descriptor to each file record.

New methods are needed to handle the following problems.

- Support for *out-of-core* computations, i.e. applications that work on very large data structures that do not fit into main memory.
- Overlapping I/O and computation.
- Support for real-time output (i.e. graphics and visualization).

Integration of Task and Data Parallelism

Data parallel languages like Vienna Fortran 90 are well suited for a wide range of scientific and numerical applications. However, many advanced engineering applications, like for example aircraft design, consist of a number of interacting coarse grained tasks. An efficient implementation of such applications requires to utilize both the parallelism inherent within each task as well as parallelism between different tasks. In order to support these kind of applications, language features and methods for task creation, synchronization, task termination, and mechanism for realizing communication between different tasks have to be provided.

Recently, Chapman et.al. [35] have proposed language extensions to Fortran 90 that attempt to integrate the data parallel and task parallel programming paradigms. They propose methods for handling task parallelism based on light weighted threads that interact by sharing access to Shared Data Abstractions (SDAs). SDAs are an extension of Fortran 90 modules and consist of data structures and methods for manipulating this data. Currently support for task parallelism along these lines is being implemented in the framework of the Vienna Fortran Compilation System.

Other work in this area includes Fortran M [45], which extends Fortran 77 with features for process management and message passing between processes via channels, and Fx [116], another Fortran extension, which provides parallel sections that allow the concurrent activation of subroutines.

Dynamic Data Distributions

Whereas the framework presented in this thesis is general enough to handle programs with dynamically distributed data objects, there remains much work to be done to provide efficient implementation for all the features provided by Vienna Fortran in the context of dynamic data distributions. In particular, new interprocedural analysis techniques that determine the set of different distributions that may reach certain array references are a prerequisite for many program optimizations. Furthermore, advanced optimization techniques with respect to communication placement and reuse of communication schedules have to be investigated.

Full Implementation

A subset of Vienna Fortran 90 has been implemented within the Vienna Fortran compilation system. In addition, a prototype runtime support system with respect to the basic intrinsic distribution functions (including alignment) has been implemented. This prototype system is currently enhanced and integrated into the Vienna Fortran compiler. Nevertheless, much work remains to be done for a full implementation of Vienna Fortran 90. In particular, new concepts and compilation techniques for an efficient implementation of dynamic data distribution, irregular distributions, and user defined distribution functions have to be developed.

Bibliography

[1] A.V. Aho, R. Sethi, J.D. Ullman. Compilers. Principles, Techniques and Tools. Series in Computer Science. Addison Wesley, 1988.

[2] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. Ph.D. Thesis, Rice University, Houston, Texas, 1983.

[3] J. R. Allen, K. Kennedy. *Automatic Translation of Fortran Programs to Vector Form*. Computer Science Technical Report, COMP TR84-9, Rice University, Houston, Texas, 1984.

[4] F. Andre, J. Pazat and H. Thomas. Pandore: A system to manage data distribution. *Proc. 1990 ACM International Conference on Supercomputing*, Amsterdam, June 1990.

[5] F. Andre, T.Priol. Programming Distributed Memory Parallel Computers with explicit message passing. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, USA, pp. 90-97, April 1992.

[6] Applied Parallel Research Inc. APR's FORGE 90 Parallelization Tools for High Performance Fortran. June 93

[7] V.Balasundaram, A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. In *Journal of Parallel and Distributed Computing*, 9, 154-170, 1990

[8] V.Balasundaram, G.Fox, K.Kennedy, U.Kremer, *An Interactive Environment for Data Partitioning and Distribution*, Fifth Distributed Memory Computing Conference, Charleston, S. Carolina, April 9-12, 1990

[9] V.Balasundaram, K.Kennedy, U.Kremer, K.McKinley, J.Sublhok, *The ParaScope editor: An interactive parallel programming tool*, Supercomputing 89, Reno, Nevada, November 89, 540-550

[10] S.Benkner, B.Chapman, H.P.Zima. Vienna Fortran90. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, USA, pp. 51-59, April 1992.

[11] S. Benkner, P. Brezany, H.P. Zima. *Functional Specification of the Prepare Parallelization Engine*. Internal Report of the Prepare Project, June 1993.

[12] S. Benkner, P. Brezany, H.P. Zima. *Program Normalization*. PREPARE Project Document, June 1993.

[13] S. Benkner, H.P. Zima. Massively Parallel Architectures and Their Programming Paradigms: Recent Developments, AICA'93, Lecce, Italy

[14] S. Benkner, P. Brezany, H.P. Zima. Compiling High Performance Fortran in the Prepare Environment. In *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993

[15] S. Benkner, P. Brezany, H. Zima. Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler. In P.A. Fritszon, editor, *Compiler Construction, Proceedings of the 5th International Conference*, LNCS 786, pages 324-338. Springer-Verlag, April 1994.

[16] R. Berrendorf, M. Gerndt, W. Nagel and J. Prümmer. SVM Fortran. Technical Report, Research Center Juelich (KFA), Germany, September 1993.

[17] M.J. Berger and S.H.Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. IEEE Trans. Comput., vol 36, no. 5, pp. 570-580, 1987.

[18] L. Bomans,D. Roose,R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming. Parallel Computing, 15:119-132,1990

[19] T. Brandes. Efficient data parallel programming without explicit message passing distribute memory multiprocessors. Technical Report AHR-92-4, GMD, 1992.

[20] L. Breebaart, E. Paalvast and H. Sips. A Rule Based Transformation System for Parallel Languages. In *Proceedings of the Third International Workshop on Compilers for Parallel Computers*, Vienna, Austria, 1992.

[21] L. Breebaart. Experiences with Rule-based Compilation. In *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, Delft University of Technology, The Netherlands, December 1993.

[22] P.Brezany, M.Gerndt, V.Sipkova, H.P.Zima. SUPERB Support for Irregular Scientific Computations. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, USA, pp. 314-321, April 1992.

[23] P.Brezany, M.Gerndt, P.Mehrotra, H.P.Zima. Concurrent File Operations in a High Performance Fortran. In *Proceedings of Supercomputing'92.* Minneapolis, USA, pp. 230-237, November 1992.

[24] P.Brezany, B.Chapman, H.Zima. Automatic Parallelization for GENESIS. Austrian Center for Parallel Computation. Technical Report ACPC/TR 92–16, November 1992.

[25] F.Bodin, L. Kervella and T. Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In *Proceedings Supercomputing'93*, pp. 274-283, Portland, Oregon November 93.

[26] M. Burke, R. Cytron. Interprocedural dependence analysis and parallelization. In *Proc. SIGPLAN '86 Symp. Compiler Construction*, June 1986, pp. 162-175.

[27] D. Callahan, K. Kennedy Compiling programs for distributed memory multiprocessors. The Journal of Supercomputing, 2(2):151-170,1988

[28] D. Callahan, K. Kennedy Analysis of interpocedural side effects in a parallel programming environment. In Proc. First International Conference on Supercomputing, Athens, Greece, July 1987

[29] S. Chatterjee, J.R. Gilbert, F. Jong, R. Schreiber, S. Teng. Generating local addresses and communication sets for data-parallel programs. In *4th ACM PPOPP*, pages 149-158. ACM, May 1993.

[30] B.Chapman, P.Mehrotra, H.P.Zima. Vienna FORTRAN - A Fortran Language Extension for Distributed Memory Multiprocessors. Technical Report, ICASE, NASA Langley Research Center, 1991.

[31] B. M. Chapman, P. Mehrotra, H. Zima: *Programming in Vienna Fortran*, Scientific Programming, Vol.1, No.1, 1992.

[32] B. M. Chapman, P. Mehrotra, H. Zima: *An Alternative Model for Distribution and Alignment in High Performance Fortran Without Templates*, Scientific Programming, Vol.1, No.1, 1992.

[33] B. Chapman, T. Fahringer, H. Zima. Automatic support for Data Distribution on Distributed Memory Multiprocessor Systems. In *Proceedings Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Oregon, August 12-14,1993

[34] B. M. Chapman, S.Benkner, R.Blasko, P.Brezany, M.Egg, T.Fahringer, H.M.Gerndt, B.Knaus, P.Kutschera, H.Moritsch, A.Schwald, V.Sipkova, H.Zima. *Vienna Fortran Compilation System. Version 1.0. User's Guide.* Institute for Software Technology and Parallel Systems, University of Vienna, January, 1993.

[35] B. Chapman, P. Mehrotra, J. Rosendale, and H. Zima. A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism. Technical Report, TR 94-1, Institute for Software Technology and Parallel Systems, University of Vienna, March 1994.

[36] B. Chapman, P. Mehrotra, H. Zima. Extending HPF For Advanced Data Parallel Applications. Technical Report, TR 94-7, University of Vienna, May 1994.

[37] M. Chen and J. Li. Optimizing Fortran 90 programs for data motion on massively parallel systems. Technical Report YALE/DCS/TR-882, Yale University, January 1992.

[38] A. L. Cheung and A. P. Reeves. The paragon multicomputer environement: A first implementation. Technical Report EE-CEG-89-9, Cornell University, Ithaca, NY, July 1989.

[39] V.Chvatal. Linear Programming. W.H.Freeman and Company, New York, 1983

[40] Cray Research, Incorporated. Cray T3D, Technical Summary.

[41] R. Das, J. Saltz. A manual for Parti runtime primitives - revision 2. *Internal Research Report*, University of Maryland, 1992.

[42] T. Fahringer. Automatic Performance Predicition for Parallel Programs on Massively Parallel Computers. Ph.D. Thesis, Institute for Software Technology and Parallel Systems, University of Vienna, September 1993.

[43] T. Fahringer and C. Huber. *The Weight Finder, A Profiler for Fortran 77 Programs*, September 1992. User Manual.

[44] T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. *In Proc. of the 7th ACM International Conference on Supercomputing 1993*, Tokyo, Japan, July 1993.

[45] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992, Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[46] G.Fox, *Domain Decomposition in Distributed and Shared Memory Environments*, International Conference on Supercomputing June 8-12, 1987, Athens, Greece Lecture Note in Computer Science 297, edited by Constantine Polychronopoulos

[47] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker. Solving Problems on Concurrent Processors. Prentice Hall, 1988.

[48] G.Fox, S.Hiranadani, K.Kennedy, C.Koelbel, U.Kremer, C.Tseng, M.Wu. Fortran D Language Specification. Rice University, Technical Report COMP TR90-141, December 1990.

[49] H.M. Gerndt. Array Distribution in SUPERB. In *Proceedings of the 3rd International Conference on Supercomputing* 1989, 164-174.

[50] H.M. Gerndt, and H.P. Zima. Optimizing Communication in SUPERB
Proc. Conpar 90 - VAPP IV, LNCS 457, 300-311

[51] H.M. Gerndt. Automatic Parallelization for Distributed-Memory Multiprocessing Systems. *Ph.D. Dissertation*, University of Bonn, 1989.

[52] H.M. Gerndt. Parallelization of Multigrid Programs in SUPERB. Technical Report, University of Vienna

[53] H.M. Gerndt. Work Distribution in Parallel Programs for Distributed Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 1991.

[54] H.M. Gerndt, *Updating Distributed Variables in Local Computations*, Concurrency: Practice and Experience, Vol. 2(3), 171-193 (September 1990)

[55] H.M. Gerndt. *Program Analysis and Transformations for Message-Passing Programs*. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, Virginia, 60–67, April 26-29, 1992

[56] M. Gupta and P. Banerjee. Automatic Data Distribution on Distributed Memory Multiprocessors. UILU-ENG-90-2248, Coordinated Science Lab., University of Illinois, 1990.

[57] M. Gupta. Automatic Data Distribution on Distributed Memory Multicomputers. Ph.D. Thesis, Coordinated Science Lab., University of Illinois, 1992.

[58] S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on distributed-Memory Machines. Technical Report, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1993.

[59] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, Amsterdam, 1977.

[60] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. To appear in the *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[61] M.W.Hall, S.Hiranandani, K.Kennedy, C.Tseng, *Interprocedural Compilation of Fortran D for MIMD distributed-memory machines.* Supercomputing'92, Minneapolis, pp. 522-534.

[62] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 1991.

[63] P. Havlak, K. Kennedy An Implementation of Interprocedural Bounded Regular Section Analysis *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991

[64] High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.0*, May 3, 1993.

[65] R.Hill. MIMDizer: A new tool for parallelization. Supercomputing Review, 3(4):26-28, April 1990

[66] S.Hiranandani, K.Kennedy, C.Tseng, *Compiler Support for Machine-Independent Parallel Programming in Fortran D.*, Rice University COMP TR91-149, Houston, March, 1991

[67] S.Hiranandani, K.Kennedy, C.Tseng, *Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*, In *Proceedings of the Supercomputing 91*, Albuquerque, Nov. 1991, 86-100

[68] S. Hiranandani, K. Kennedy and C.-W. Tseng. *Compiling Fortran D for MIMD Distributed-Memory Machines.* Comm. ACM Vol. 35, No.8, 66–80, August 1992.

[69] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems, 12(1):26–60, January 1990.

[70] J. Hulman, S.Andel, B. Chapman, and H. Zima. Intelligent Parallelization within the Vienna Fortran Compilation System. In *Proceeding of the Fourth International Workshop on Compilers for Parallel Computers*, pp. 455 - 467, Delft University of Technology, The Netherlands, December 1993

[71] K. Ikudome, G. Fox, A. Kolawa and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. *Proc. 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990

[72] ISO. Fortran 90 Standard, May 1991, ISO/IEC 1539 :1991 (E)

[73] H. F. Jordan. *Scalability of Data Transport.* In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, Virginia, 1–8, April 26-29, 1992.

[74] J.B. Kam, J.D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*,7(3):305-317, July 1977

[75] A.H. Karp: *Programming for Parallelism*, Computer 20(5), 43–57, May 1987

[76] D.E. Knuth. The Art of Computer Programming, Vol. 2, Addison-Wesley, 1981

[77] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programminging*, 16(5):365–382, 1987.

[78] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186, March 1990.

[79] C. Koelbel. Compiling Programs for Nonshared Memory Machines. *Ph.D. Dissertation*, Purdue University, West Lafayette, IN, November 1990.

[80] C. Koelbel, P. Mehrotra. Compiler Support for Unstructured Scientific Computations. Rice COMP TR90-144, December, 1990.

[81] C. Koelbel. Compile time Generation of Regular communications Patterns. In *Proceedings Supercomputing 91*, Albuquerque, 101-110.

[82] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Sytems*, October 1991.

[83] D.J. Kuck, R.H. Kuhn, B.R. Leasure and M.J. Wolfe. The structure of an advanced retargetable vetorizer. In K. Hwang, editor, *Supercomputers: Design and Applications Tutorial* pp. 967-974, IEEE Society Press, Silver Spring MD

[84] H.W.Kuhn. Solvability and Consistency for Linear Equations and Inequalities. American Mathematical Monthly 63 (1956), pp.217-232

[85] J. Li and M. Chen. Index domain alignment:minimizing cost of cross-reference between distributed arrays. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, October, 1990.

[86] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. In *Proceedings of Supercomputing '89*, New York, NY, November 1990.

[87] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems* Vol.2(3), 361-376, July 1991.

[88] J. Li. Compiling Crystal for Distributed-Memory Machines. PhD Thesis, Yale University, October 1991

[89] Z. Li, P.C. Yew, Efficient interprocedural analysis for program parallelization and restructuring. In *Proc. ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, New Haven, CT, pages 85-99, July 1988

[90] D. Loveman. High Performance Fortran Proposal. DEC, Massively Parallel Systems Group, Maynard, MA, 1992.

[91] J. McGraw, S. Skedzielewski, S. Allan, R. Oldenhoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Language reference manual. Report M-146, Lawrence Livermore National Laboratory, March 1985.

[92] P. Mehrotra, J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, Vol. 5, 339-361, 1987.

[93] P. Mehrotra, J. Van Rosendale. Programming Distributed Memory Architectures Using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[94] J. H. Merlin. *ADAPTing FORTRAN 90 Array Programs for Distributed Memory Architectures*, In *Proceedings of the First International Conference of the ACPC*, September 1991, Salzburg, Austria.

[95] J. H. Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. Technical Report, SNARC 92-02, University of Southampton, 1992

[96] M. Metcalf and C.Reid. Fortran 8x Explained, Oxford University Press, 1987.

[97] M. Metcalf and C.Reid. Fortran 90 Explained, Oxford University Press, 1990.

[98] MIMDizer user's guide, version 7.02. Technical report, Pacific Sierra Research Corporation, 1991.

[99] E. Paalvast, H. Sips, and A. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 124-131, 1991.

[100] E. Paalvast and H.Sips. Booster: a High-level Language for Portable Parallel Algorithms. Applied Mathematics 8, pp.177-192, 1991.

[101] E. Paalvast. Programming for Parallelism and Compiling for Efficiency. Ph.D. Thesis, Delft University of Technology, 1992.

[102] C. Pancake and D. Bergmark. Do Parallel Languages Respond to the Need of Scientific Programmers. IEEE Computer, 23(12):13-23, December 1990.

[103] D.M. Pase, T. MacDonald, A. Meltzer. MPP Fortran Programming Model. Technical Report, Cray Research, March 1994.

[104] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory machines. TR-90-09-03, Dept. Electrical and Comp. Engenieering, Lousiana State University, September 1990.

[105] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 1–999. ACM SIGPLAN, June 1989.

[106] L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. In *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers.* Delft University of Technology, The Netherlands, 1993

[107] M. Rosing, R. W. Schnabel, and R. P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

[108] R. Ruehl, M. Annaratone. Parallelization of Fortran Code on Distributed-Memory Parallel Processors. In *Proceedings of the 4th International Conference on Supercomputing 1990*, Amsterdam, 342-353.

[109] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. Report 90-59, ICASE, 1990.

[110] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[111] J. Saltz, R. Das, R. Ponnusamy, D. Mavriplis, H. Berryman and J. Wu. Parti Procedures for Realistic Loops. In *Proceedings of DMCC6*, Portland, Oregon, 1991.

[112] R. Das, J. Saltz, H. Berryman. A manual for Parti runtime primitives - revision 1. *Interim Report 91-17*, ICASE, 1991.

[113] H.J. Sips. Local Set Generation Revisited. Technical Report, Advanced School of Computing and Imaging, Delft University of Technology, Delft, the Netherlands, January 1994.

[114] G. Steele. Proposals for amending high performance Fortran. *Communicated at the High Performance Forum*, Houston, January 1992.

[115] J.M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report, CMU-CS-93-109, Carnegie Mellon University, February 1993.

[116] J. Subhlok and T.Gross. Task Parallel Programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science. Carnigie Mellon University, Pittsburgh, PA.

[117] Thinking Machines Corporation. CM Fortran Reference Manual, Version 5.2. Thinking Machines, Cambridge, MA, September 1989.

[118] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary Thinking Machines, Cambridge, MA, October 1991

[119] R. Triolet, F. Irigion, P. Feautrier, Direct Parallelization of CALL Statements In *Proc. SIGPLAN'86 Symposium on Compiler Construction*, pages 176-185, July 1986

[120] C.W. Tseng. *An optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[121] P. S. Tseng. A systolic array programming language. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1125–1130, April 1990.

[122] PREPARE Document: Towards the User Interface Design, Steria S.A., Feb. 1993.

[123] PREPARE Document: Towards the PREPARE Abstract Machine, Parsytec Computer, Feb. 1993.

[124] PREPARE Document: The Common Computation Model for the PREPARE Parallelization Engine (Draft), IRISA, January 11, 1993.

[125] PREPARE Document: Specification of a Case Study Compiler using the Cocktail Toolbox, IRISA, March 1993.

[126] PREPARE Document: D1.1/1 Specification of the Input Language Subset, ACE, March 1993.

[127] M. Weiser. Programmers use Slices When Debugging. Commun. ACM, Vol. 25, July 1982, 446–452.

[128] M. Weiser. Program Slicing. IEEE Transactions on Softw. Eng., Vol. 10, No. 4, July 1984, 352–357.

[129] B. Wender. Performance Measurement of Parallel Fortran Programs in the Vienna Fortran Compilation System. Technical report, University of Vienna, Institute for Software Technology and Parallel Systems, July 1994.

[130] M. Wolfe. Optimizing Supercompilers for Supercomputing. The MIT Press, Cambridge, Massachusetts, 1989.

[131] M. Wolfe. New Program Restructuring Technology. In: H. Zima (ed.). *Parallel Computation*, LNCS 591, Springer-Verlag, Berlin, 1992, 13-36.

[132] J. Wu, Joel Saltz, Harry Berryman, Seema Hiranandani. Distributed Memory Compiler Design For Sparse Problems. ICASE Report No. 91-13, January 1991.

[133] M. Wu and G. Fox. A Test Suite Approach for Fortran90D Compilers on MIMD Distributed Memory Parallel Computers. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, USA, pp. 393-400, April 1992.

[134] H. Zima. Compilerbau I: Analyse, 2nd (revised) edn., Reihe Informatik, Band 36, Bibliographisches Institut, Mannheim, 1989.

[135] H. Zima. Compilerbau II: Synthese und Optimierung, Reihe Informatik Band 37, Bibliographisches Institut, Mannheim, 1983.

[136] H. Zima,H.-J. Bast, M. Gerndt, and P. Hoppen. Semi-automatic parallelization of Fortran programs. In *CONPAR 86. Conf. on Algorithms and Hardware for parallel processing* (W. Händler, D. Haupt, K. Leusch, W. Juling, and O. Lange, eds.), Lecture Notes in Computer Science 237, pp.287-94, Springer, Berlin.

[137] H. Zima, H.-J. Bast, and H.M. Gerndt. SUPERB - a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6, 1-18, 1988.

[138] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.

[139] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[140] H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. In: *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, February 1993.

Appendix A

Syntax

The notation used in the syntax rules is a variation of Backus-Naur form and in accordance with the Fortran 90 standard document [72].

The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[ ]	encloses an optional item
[ ] ...	encloses an optionally repeated item which may occur zero or more times

Assumed syntax rules are:

xyz-list	**is**	*xyz* [, *xyz*] ...
xyz-name	**is**	*name*
integer-xyz	**is**	*xyz*

The following nonterminals are taken from the Fortran 90 standard document [72].

assumed-shape-spec-list	*int-expr*
array-element	*object-name*
attr-spec	*section-subscript*
component-def-stmt	*stat-variable*
executable-construct	*structure-component*
explicit-shape-spec-list	*subscript*
entity-decl-list	*subscript-triplet*
expr	*specification part*
data-ref	*type-spec*
dummy-arg-name-list	*variable-name*

A **constraint** is a restriction on the syntax rules that is capable of being checked by the compiler.

A.1 Processor Array Declarations

proc-decl-stmt	**is**	`PROCESSORS` [[`,` *proc-attr-spec-list*] `::`] *proc-decl-list*
proc-decl	**is**	*proc-array-name* `(` *proc-shape-spec-list* `)`
proc-attr-spec	**is**	`DIMENSION` `(` *proc-shape-spec-list* `)`
proc-array-name	**is**	*object-name*
proc-shape-spec-list	**is**	*explicit-shape-spec-list*

A.2 Processor Array References

proc-ref	**is**	*proc-array-name* [*dimension-permutation*]
	or	*proc-array-section* [*dimension-permutation*]
	or	*proc-array-element*
proc-array-section	**is**	*proc-array-name* `(` *section-subscript-list* `)`
proc-array-element	**is**	*array-element*
dimension-permutation	**is**	`(/` *int-const-expr-list* `/)`
section-subscript	**is**	*subscript*
	or	*subscript-triplet*
	or	*vector-subscript*

A.3 Type Declaration Statement

vf-type-declaration-stmt	**is**	*type-spec* [[`,`*vf-attr-spec*] ... `::`] *entity-decl-list*
vf-attr-spec	**is**	*attr-spec*
	or	*dist-attr-spec*
	or	*align-attr-spec*
	or	*dynamic-attr-spec*
	or	*location-attr-spec*
	or	`RESTORE`

Constraint: If *dist-attr-spec, align-attr-spec, dynamic-attr-spec, location-attr-spec* or the *RESTORE* attribute is present, *type-spec* must not be *CHARACTER*.

A.4 The DISTRIBUTED Attribute

dist-attr-spec	**is**	`DISTRIBUTED` *dist-spec*
	or	`DISTRIBUTED TO` *proc-ref*

dist-spec	**is**	(*dist-type-spec*) [`TO` *proc-ref*]
	or	*assumed-dist-spec*
dist-type-spec	**is**	*composite-dist-type-spec*
	or	*general-dist-type-spec*
composite-dist-type-spec	**is**	*dim-dist-type-spec-list*
dim-dist-type-spec	**is**	*dim-dist-func-ref*
	or	*dist-extraction*
	or	:
general-dist-type-spec	**is**	*dist-func-ref*
	or	*dist-extraction*
	or	*dist-type-name*
dim-dist-func-ref	**is**	*dist-func-ref*
dist-func-ref	**is**	`BLOCK`[(*int-expr*)]
	or	`CYCLIC`[(*int-expr*)]
	or	`S_BLOCK`(*expr*)
	or	`B_BLOCK`(*expr*)
	or	`INDIRECT`(*expr*)
	or	*mapping-function-ref*
	or	*dist-type-name*
dist-extraction	**is**	=*variable-name*[.*dimension-qualifier*]
	or	=*structure-component*[.*dimension-qualifier*]
dimension-qualifier	**is**	*int-expr*
assumed-dist-spec	**is**	(*)

Constraint: The number of *dim-dist-type-spec*s must match the rank of the array.

Constraint: An *assumed-dist-spec* may be specified for statically distributed dummy arrays only.

Constraint: A distribution-extraction occurring in the *DISTRIBUTED* attribute of a statically distributed array must refer to a another statically distributed array.

Constraint: If a *proc-ref* is not specified, the primary processor array is substituted by default.

Constraint: Any expressions occurring in a *dist-attr-spec* must be specification expressions.

Constraint: The argument of *S_BLOCK* and *B_BLOCK* must be a one-dimensional integer array expression.

Constraint: The argument of *INDIRECT* must be an integer array expression.

Constraint: In a structure-component occurring in a *dist-extraction* there may be more than one *part-ref* with non-zero rank if and only if the right-most *part-ref* does not have the *DYNAMIC* attribute.

Constraint: A *dimension-qualifier* must be an integer constant expression.

A.5 The ALIGNED Attribute

align-attr-spec	**is**	`ALIGNED (` *align-spec* `)`
align-spec	**is**	*element-align-spec*
	or	*section-align-spec*
element-align-spec	**is**	*alignee-spec* `WITH` *align-source-spec*
alignee-spec	**is**	[*alignee-name*] `(` *dim-alignee-spec-list* `)`
	or	*align-function-ref*
dim-alignee-spec	**is**	*bound-variable-name*
	or	`:`
align-source-spec	**is**	*align-source-object* [`(` *source-subscript-list* `)`]
source-subscript	**is**	*subscript*
	or	*subscript-triplet*
	or	*align-subscript*
section-align-spec	**is**	[[*alignee-name*] `WITH`] *align-source-section*
align-source-section	**is**	*variable-name*
	or	*structure-component*
alignee-name	**is**	*variable-name*
	or	*structure-component*
align-source-object	**is**	*variable-name*
	or	*structure-component*
align-function-ref	**is**	*mapping-function-ref*

Constraint: The number of *dim-alignee-specs* must match the rank of the alignee.

Constraint: A *bound-variable* occurring in an *align-spec* is implicitly declared of type integer. Any variable in the corresponding scoping unit with the same name is unaffect by it. The scope of the bound variable is restricted to the *align-spec*.

Constraint: A statically allocated array must not be aligned with a dynamically allocated array.

Constraint: *subscript* and *subscript-triplet* must not contain a bound variable.

Constraint: If *alignee-name* is not present, the *ALIGNED* attribute is applied to all arrays in the *entity-decl-list*.

Constraint: The *align-source-section* must have the same rank as the *alignee*.

Constraint: All expressions occurring in an *align-spec* must be specification expressions.

Constraint: If an *alignee-name* or a *align-source-object* is a *structure-component*, then there may be more than one *part-ref* with non-zero rank if and only if the right-most *part-ref* does not have the *DYNAMIC* attribute.

A.6 Mapping Procedures

mapping-subprogram	**is**	*mapping-function-stmt* *mapping-specification-part* *mapping-execution-part* *end-mapping-function-stmt*
mapping-stmt	**is**	`MAPPING` *mapping-name* [(*dummy-arg-name-list*)]
end-mapping-function-stmt	**is**	`END` [`MAPPING` [*mapping-function-name*]]
mapping-specification-part	**is**	*mapping-specification* [*specification part*]
mapping-specification	**is**	`FROM` *from-domain-decl* `TO` *to-domain-decl*
from-domain-decl	**is**	*from-domain-name* [(*assumed-shape-spec-list*)]
to-domain-decl	**is**	*to-domain-name* [(*assumed-shape-spec-list*)]
mapping-execution-part	**is**	*mapping-exec-construct* [*mapping-exec-construct*] ...
mapping-exec-construct	**is**	*index-mapping-stmt*
	or	*executable-construct*
index-mapping-stmt	**is**	*from-domain-ref* `->` *to-domain-ref*
	or	*from-domain-ref* `MAP(`*mapping-type-spec*`)` `TO` *to-domain-ref*
from-domain-ref	**is**	*data-ref*
to-domain-ref	**is**	*data-ref*
map-type-spec	**is**	*composite-mapping*
	or	*general-mapping*
composite-mapping	**is**	*dimensional-mapping-list*

dimensional-mapping	**is**	*mapping-function-ref*
	or	`BLOCK[(`*int-expr*`)]`
	or	`CYCLIC[(`*int-expr*`)]`
	or	`S_BLOCK(`*expr*`)`
	or	`B_BLOCK(`*expr*`)`
	or	`INDIRECT(`*expr*`)`
	or	*dist-type-name*
general-mapping	**is**	*mapping-function-ref*

Constraint: *from-domain-name* and *to-domain-name* may only be referenced in the *mapping-specification* and *index-mapping-stmt*.

Constraint: All elements of the *from-domain* must be referenced at least once.

Constraint: An *executable-construct* must not be an action-stmt that is related with I/O operations (i.e. backspace, close, endfile, inquire, open, print, read, rewind, write).

Constraint: A *mapping-function-ref* occurring in a *dimensional-mapping* must be a 1-1 mapping function.

Constraint: The argument of *S_BLOCK* and *B_BLOCK* must be a one-dimensional integer array expression.

Constraint: The argument of *INDIRECT* must be an integer array expression.

A.7 Dynamically Distributed Arrays

dynamic-attr-spec	**is**	`DYNAMIC`
	or	`RANGE (` *dist-range-spec-list* `)`
	or	`CONNECT (` *connection* `)`
	or	*initial-dist*
connection	**is**	*dist-extraction*
	or	*align-spec*
dist-range-spec	**is**	*dist-spec*
	or	*dist-range-extraction*
initial-dist	**is**	*dist-attr-spec*
	or	*align-attr-spec*
dist-range-extraction	**is**	*dist-extraction*

Constraint: The *RANGE* attribute may be specified only for arrays with the DYNAMIC attribute, or for arrays with an *assumed* distribution specification.

Constraint: If the *CONNECT* attribute is specified, the *RANGE* attribute must not be specified.

Constraint: If the *CONNECT* attribute is specified, the *DYNAMIC* attribute must be specified too.

Constraint: If the *CONNECT* attribute is specified, an initial distribution or a distribution range must not be specified.

Constraint: The array in the *connection* must have the *DYNAMIC* attribute.

Constraint: If an initial distribution is specified by means of distribution extraction, it must refer to a statically distributed array or to a dynamically distributed (primary) array for which an initial distribution has been specified.

Constraint: Any expression occurring in a *dist-range-spec* or *connection* must be a specification expression.

A.8 The DISTRIBUTE Statement

distribute-stmt	**is**	`DISTRIBUTE (` *dist-list* [`,`*notransfer*] [`, STAT =` *stat-variable*] `)`
dist	**is**	*distribute-object dist-spec*
distribute-object	**is**	*variable-name*
	or	*structure-component*
notransfer	**is**	`NOTRANSFER` [`(` *notransfer-object-list* `)`]
notransfer-object	**is**	*variable-name*
	or	*structure-component*

Constraint: A distribution extraction occurring in a *dist-spec* must not refer to any other array of the same *DISTRIBUTE* statement.

Constraint: If a distribute-object or a notransfer-object is a structure component, more than one part-ref may have non-zero rank.

Constraint: *distribute-object* must have the *DYNAMIC* attribute specified but not the *CONNECT* attribute.

Constraint: All arrays appearing in the *notransfer-object-list* must be contained in the connect set of one of the objects in the *dist-list*.

A.9 The ALIGN Statement

align-stmt	**is**	`ALIGN (` *align-list* [`,` *notransfer*] [`, STAT =` *stat-variable*] `)`
align	**is**	*align-object* `(` *align-spec* `)`
align-object	**is**	*variable-name*
	or	*structure-component*
notransfer	**is**	`NOTRANSFER` [`(` *notransfer-object-list* `)`]
notransfer-object	**is**	*variable-name*
	or	*structure-component*

Constraint: An object appearing in the *align-list* must not have any other object connected to it.

Constraint: If an align-object is a structure component, more than one part-ref may have non-zero rank.

Constraint: An *align-object* must have the *DYNAMIC* attribute.

Constraint: An *align-object* must not have any other objects connected to it.

A.10 The ALLOCATE Statement

allocate-stmt	**is**	`ALLOCATE (` *allocation-list* [`, STAT =` *stat-variable*] `)`
allocation	**is**	*allocate-object* [`(` *explicit-shape-spec-list* `)`] [*da-spec*]
allocate-object	**is**	*variable-name*
	or	*structure-component*
da-spec	**is**	*dist-spec*
	or	`(` *align-spec* `)`

Constraint: If *da-spec* is specified, the corresponding *allocate-object* must have the *DYNAMIC* attribute.

Constraint: If *da-spec* is specified, the corresponding *allocate-object* must not have the *CONNECT* attribute.

A.11 Derived Types

dist-derived-type-def	**is**	*vf-derived-type-stmt* *component-def-stmt* [*component-def-stmt*] ... *end-type-stmt*
vf-derived-type-stmt	**is**	`TYPE` [`::`] *type-name* [, `PROCESSORS` [`::`] *proc-decl*]
component-def-stmt	**is**	*type-spec* [[, *vf-comp-attr-spec*] ... `::`] *component-decl-list*
vf-comp-attr-spec	**is**	*component-attr-spec*
	or	*dist-attr-spec*
	or	*align-attr-spec*
	or	*dynamic-attr-spec*
	or	*location-attr-spec*

Constraint: The *DISTRIBUTED*, *ALIGNED*, *DYNAMIC* or *CONNECT* attribute must not be specified if *type-spec* designates a distributed derived type.

Constraint: If both the *DYNAMIC* and *POINTER* attributes are specified, an *initial-distribution* must not be specified.

Constraint: If both the *DYNAMIC* and *POINTER* attributes are specified, the *CONNECT* attribute must not be specified.

Constraint: A *proc-shape-spec* occurring in a *vf-derived-type-stmt* must be an *assumed-shape-spec*.

Constraint: If *location-attr-spec* is present, *type-spec* must designate a derived type.

Constraint: If em dist-attr-spec, align-attr-spec, dynamic-attr-spec, or *location-attr-spec* is present, *type-spec* must not be *CHARACTER*

The syntax for referencing structure components as defined by the Fortran 90 standard [72] is included here for sake of consistency.

structure-component	**is**	*data-ref*
data-ref	**is**	*part-ref* [% *part-ref*] ...
part-ref	**is**	*part-name* [(*section-subscript-list*)]

Constraint: In a *data-ref*, there must not be more than one *part-ref* with nonzero rank. A *part-name* to the right of a *part-ref* with nonzero rank must not have the *POINTER* attribute.

Constraint: In a *structure-component* occurring in a *dist-extraction*, a *dist-spec*, or in an *align-spec* there may be more than one *part-ref* with non-zero rank if and only if the right-most *part-ref* does not have the *DYNAMIC* attribute.

A.12 The LOCATION Attribute

location-attr-spec	**is**	`LOCATION` (*proc-ref*)

Constraint: If *location-attr-spec* is present, the corresponding *type-spec* must designate a derived type.

Constraint: An entity which has the *LOCATION* attribute must not have the *DISTRIBUTED*, *ALIGNED*, or *DYNAMIC* attribute.

A.13 Distribution Type Definitions

dist-type-def	**is**	`DTYPE` *dist-type-name* [(*dummy-arg-name-list*)] [*type-declaration-stmt*] ... [*dist-type-name* =] (*dist-type-spec*) *end-dist-type-stmt*
end-dist-type-stmt	**is**	`END DTYPE` [*dist-type-name*]
dist-type-ref	**is**	*dist-type-name* [(*actual-arg-spec-list*)]

A.14 Forall

forall-loop	**is**	*forall-statement* [*private-var-decl-list*] [*forall-block*] *end-forall*
forall-statement	**is**	*label-forall-statement*
	or	*nonlabel-forall-statement*
nonlabel-forall-statement	**is**	[*forall-construct-name* \`\`:''] `FORALL` *forall-control*
forall-control	**is**	(*control-variable-list*) [*work-dist-spec*]
control-variable	**is**	*variable-name* = *int-expr* , *int-expr* , *int-expr*
work-dist-spec	**is**	*on-clause*
	or	`DISTRIBUTED` *dist-spec*
on-clause	**is**	`ON` *proc-array-element*
	or	`ON OWNER` (*array-element*)
end-forall	**is**	*end-forall-statement*

A.15 The DCASE Construct

dcase-construct	**is**	*select-dcase-stmt* [*dcase-stmt* *block*] ... *end-select-stmt*
select-dcase-stmt	**is**	[*dcase-construct-name* :] `SELECT DCASE` (*array-name-list*)
dcase-stmt	**is**	`CASE` *dcase-selector*
dcase-selector	**is**	*dcase-dist-spec-list*
	or	`DEFAULT`
dcase-dist-spec	**is**	[*name-tag* :] *dist-spec*
	is	[*name-tag* :] *
name-tag	**is**	*variable-name*

Constraint: A *dist-spec* must not contain a *distribution-extraction*.

Constraint: If a *name-tag* is specified, the corresponding array must appear in the *select-dcase-stmt*.

Constraint: All *name-tag*s appearing within a single *dcase-selector* must be different.

A.16 The DISTRIBUTED, ALIGNED, DYNAMIC, CONNECT, LOCATION, RANGE, RESTORE Statements

distributed-stmt	**is**	`DISTRIBUTED` *dist-spec* [::] *entity-decl-list*
	or	`DISTRIBUTED TO` [::] *entity-decl-list*
aligned-stmt	**is**	`ALIGNED (` *align-spec* `)` [::] *entity-decl-list*
dynamic-stmt	**is**	`DYNAMIC` [::] *entity-decl-list*
connect-stmt	**is**	`CONNECT (` *connection* `)` [::] *entity-decl-list*
location-stmt	**is**	`LOCATION (` *proc-ref* `)` [::] *entity-decl-list*
range-stmt	**is**	`RANGE (` *dist-range-spec-list* `)` [::] *entity-decl-list*
restore-stmt	**is**	`RESTORE` [::] *entity-decl-list*

Appendix B

Intrinsic Functions

In this subsection we summarize the intrinsic functions provided by Vienna Fortran 90. We present here only those intrinsic functions which are not in Fortran 90 or Fortran 90 intrinsics which have been modified.

B.1 Intrinsic Distribution Function

B.1.1 BLOCK(M)

Optional Argument. `M`

Description. Determines a block distribution.

Class. 1-1 distribution function.

Argument. M must be scalar of type integer.

Result Type and Value. See page 27.

B.1.2 B_BLOCK(ARRAY)

Description. Determines a general block distribution.

Class. 1-1 distribution function.

Argument. `ARRAY` must be a rank one array of type integer.

Result Type and Value. See page 29.

B.1.3 CYCLIC(M)

Optional Argument. `M`

Description. Determines a block distribution.

Class. $1-1$ distribution function.

Argument. M must be scalar of type integer.

Result Type and Value. See page 31.

B.1.4 INDIRECT(ARRAY)

Description. Determines a indirect distribution.

Class. $n - m$ distribution function.

Argument. `ARRAY` must be a rank n array of type integer.

Result Type and Value. See page 35.

B.1.5 S_BLOCK(ARRAY)

Description. Determines a general block distribution.

Class. 1-1 distribution function.

Argument. `ARRAY` must be a rank one array of type integer.

Result Type and Value. See page 29.

B.2 Distribution Inquiry Functions

B.2.1 DISTRIBUTED(SOURCE)

Description. Determines whether an array or scalar has a defined distribution.

Class. Distribution Inquiry function.

Argument. `SOURCE` may be of any type and may be array valued or scalar.

Result Type. Logical scalar.

Result Value. The result has the value true if `SOURCE` has a defined distribution, otherwise the value false.

B.2.2 DYNAMIC(SOURCE)

Description. Determines whether an array or a scalar is dynamically distributed.

Class. Distribution Inquiry function.

Argument. `SOURCE` may be of any type and may be array valued or scalar.

Result Type. Logical scalar.

Result Value. The result has the value true if `SOURCE` is dynamically distributed, otherwise the value false.

B.2.3 OWNED(A, PROCESSOR)

Optional Argument. PROCESSOR

Description. Determines whether A is owned by a processor.

Class. Elemental distribution inquiry function.

Arguments.

A	may be of any type.
PROCESSOR (optional)	must be scalar and of type integer with a value in the range $1 \leq$ PROCESSOR $\leq$ \$NP. If it is omitted, it is if it were present with the value \$MY_PROC.

Result Type. Logical.

Result Value. The result is true if A is owned by PROCESSOR.

B.2.4 OWNER(A, PROCESSOR)

Optional Argument. PROCESSOR

Description. Determines processor that owns A.

Class. Distribution inquiry function.

Arguments.

A	must be a scalar of any type.
PROCESSOR (optional)	must be scalar and of type integer with a value in the range $1 \leq$ PROCESSOR $\leq$ \$NP. If it is omitted, it is if it were present with the value \$MY_PROC.

Result Type. The result is scalar of type integer with a value in the range $1 \leq$ OWNER(E,PROCESSOR) $\leq$ \$NP.

Result Value. The result is the number of the processor that owns A. If more than one processor owns A, a system dependent value is returned.

B.2.5 OWNERS(A)

Description. Determines all processors that own A.

Class. Distribution inquiry function.

Argument. A must be a scalar of any type.

Result Type and Shape. The result is a rank one array of type integer with size equal to the number of processors that own A.

Result Value. Each element P of the result is a unique value in the range $1 \leq$ P $\leq$ \$NP such that OWNED(A,P) is true.

B.3 Processor Inquiry Functions

B.3.1 $NP

Description. Determines the number of processors executing a program.

Class. Processor inquiry function.

Result Type. Scalar integer.

Result Value. Number of processors executing a program.

B.3.2 $MY_PROC

Description. Determines the number of the executing processor.

Class. Processor inquiry function.

Result Type. Scalar integer.

Result Value. The number of the executing processor, $1 \leq$ `$MY_PROC` $\leq$ `$NP`.

B.4 Extended Fortran 90 Intrinsics

B.4.1 LBOUND(ARRAY,DIM,PROCESSOR)

Optional Arguments. `DIM, PROCESSORS`

Description. Returns all the lower bounds or a specified lower bound of an array on a specific processor.

Class. Inquiry Function.

Arguments.

DIM (optional)	as in Fortran 90.
PROCESSOR (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `PROCESSOR` $\leq$ `$NP`. If it is omitted, it is if it were present with the value `$MY_PROC`.

Result Type and Shape. As in Fortran 90.

Result Value.

Case (i): If `PROCESSOR` is omitted, then as in Fortran 90.

Case (ii): If `PROCESSOR` and `DIM` is present, then the function returns the smallest index i_{DIM} such that element `ARRAY(`$i_1, \ldots, i_{\mathtt{DIM}}$`)` is owned by `PROCESSOR`.

Case (iii): If `PROCESSOR` is present but not `DIM`, then the function returns a rank one array of size n where the i-th component, $1 \leq i \leq n$, is equal to `LBOUND(ARRAY,` i`, PROCESSORS)`, where n is the rank of `ARRAY`.

B.4.2 UBOUND(ARRAY,DIM,PROCESSOR)

Optional Arguments. `DIM, PROCESSORS`

Description. Returns all the lower bounds or a specified lower bound of an array on a specific processor.

Class. Inquiry Function.

Arguments.

DIM (optional)	as in Fortran 90.
PROCESSOR (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `PROCESSOR` $\leq$ `$NP`. If it is omitted, it is if it were present with the value `$MY_PROC`.

Result Type and Shape. As in Fortran 90.

Result Value.

Case (i): If `PROCESSOR` is omitted, then as in Fortran 90.

Case (ii): If `PROCESSOR` and `DIM` is present, then the function returns the largest index i_{DIM} such that element `ARRAY(`$i_1, \ldots, i_{\mathtt{DIM}}$`)` is owned by `PROCESSOR`.

Case (iii): If `PROCESSOR` is present but not `DIM`, then the function returns a rank one array of size n where the i-th component, $1 \leq i \leq n$, is equal to `UBOUND(ARRAY,` i`, PROCESSORS)`, where n is the rank of `ARRAY`.

B.4.3 SIZE(ARRAY,DIM,PROCESSOR)

Optional Arguments. `DIM, PROCESSORS`

Description. Returns the total number of elements along a specified dimension or the total number of an array (on a specific processor).

Class. Inquiry Function.

Arguments.

DIM (optional)	as in Fortran 90.
PROCESSOR (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `PROCESSOR` $\leq$ `$NP`. If it is omitted, it is if it were present with the value `$MY_PROC`.

Result Type and Shape. As in Fortran 90.

Result Value.

Case (i): If `PROCESSOR` is omitted, then as in Fortran 90.

Case (ii): If `PROCESSOR` and `DIM` is present, then the function returns the total number of elements `ARRAY(`$i_1, \ldots, i_{\mathtt{DIM}}$`)` that are owned by `PROCESSOR`.

Case (iii): If `PROCESSOR` is present but not `DIM`, then the function returns the total number of elements of `ARRAY` that are owned by `PROCESSOR`.