Language Constructs for Data Partitioning and Distribution

P. CROOKS AND R. H. PERROTT

Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland

ABSTRACT

This article presents a survey of language features for distributed memory multiprocessor systems (DMMs), in particular, systems that provide features for data partitioning and distribution. In these systems the programmer is freed from consideration of the low-level details of the target architecture in that there is no need to program explicit processes or specify interprocess communication. Programs are written according to the shared memory programming paradigm but the programmer is required to specify, by means of directives, additional syntax or interactive methods, how the data of the program are decomposed and distributed.

1. INTRODUCTION

One solution to the need for higher-performance computers is to connect multiple sequential processors, each having its own local memory, into what is known as a distributed memory multiprocessor (DMM). The combined computational power of these processors, which communicate by passing messages between one another, may then be brought to bear on a single problem. In many cases these systems are constructed from ordinary production microprocessors; for example, the Intel iPSC/2 consists of multiple "nodes," each of which includes an Intel 80386 CPU and an 80387 FPU coprocessor. DMMs can be both cost-effective and potentially highly scalable, due to the low cost of their component microprocessors and the modular nature of their interconnection: furthermore, they can achieve high levels of performance for certain types of application.

Unfortunately programs for these machines are much more difficult to write, debug, maintain, and understand than sequential programs, being complicated by such concerns as livelock, deadlock, processor topology, communications, synchronization, task granularity, and separate address spaces. Message-passing languages, such as Occam for the Inmos transputer, offer a relatively low-level programming interface to the multiprocessing hardware; the situation is analogous to programming a sequential processor in assembly language. A further problem is that the low-level nature of a message-passing language leads to programs that are closely tied to the hardware characteristics of the DMM for which it was designed, resulting in a lack of code portability between the various DMM machines now available.

Consequently a considerable amount of current research is aimed at providing appropriate programming tools for DMMs. Included in this research is the construction of compilation systems

Received December 1993
Revised October 1994

e-mail: p. crooks or r. perrott@qub.ac.uk

The authors apologize in advance to any of the designers of the described systems for any misinterpretation or misrepresentation of their work; it certainly was not intentional.

Scientific Programming, Vol. 4, pp. 59–85 (1995)
CCC 1058-9244/95/020059-27

for translating high-level programs into message-passing code. One method of exploiting the parallelism offered by DMMs entails the decomposition (or partitioning) of data for distribution over the processors of the machine to achieve program speed-up through data-parallel execution. The parallelization strategies of a number of compilation systems based on this principle are considered in the next section.

The choice of data partition is important as it, along with the data dependencies present in the program, determines the amount of communication required between processors. This, in turn, influences the overall performance because off-processor references can be an order of magnitude more costly that references to local memory. The choice of an "optimal" data partition must take into account the program structure, compiler capabilities, characteristics of the underlying machine (memory structure, number of processors and their topology, communication characteristics), and the sizes of distributed data structures.

An appropriate heuristic method for automatically determining an optimal data partition has yet to be found. One method of overcoming this problem is to enlist the help of the user, who must then provide the system with a suitable data partition, specified by means of directives, language extensions (additional syntax), or interactive methods. Typically an iterative, experimental approach would be adopted in choosing a partition. There are many degrees of freedom in this choice but the user would normally be sufficiently au fait with the computational code to have a good idea about which partitions are the most promising (although he/she might not be so knowledgeable about the underlying hardware characteristics). Efficient parallelization may also require the help of the user, via assertions, directives, etc., with regard to global, high-level properties of the algorithm whose detection by even the most able systems may be intractable. One example of this is the specification of FORALL "loops" to indicate the possible parallel execution of loop iterations.

This article considers some of the most significant of these compilation systems. These systems provide what may be called a virtual shared memory, in other words they enable the programmer to write programs as though the memory of the target machine were a single, shared memory; this (logical) shared memory model is put into effect on the underlying (physical) distributed memory of the target DMM by the compilation system.

One example of this approach is high-performance Fortran [HPF; 1, 2] in which compiler directives are used within a Fortran 90 program to specify data distribution and redistribution. However, this survey concentrates on systems that preceded HPF and so represents the research context in which the HPF effort was established. Furthermore, HPF currently exists largely as a proposal, whereas the systems presented below have been fully (or largely) implemented.

2. DATA PARTITIONING AND DISTRIBUTION SCHEMES

One of the problems in this area is the wide range of terminology. As a consequence the following terms, as used in this article, perhaps require clarification. The terms *user* and *programmer* are used interchangeably; normally the user of the parallelization system will be the author of the program to be parallized; in any case, the use of all but one (SUPERB) of the systems covered in this section entails additional programming, thereby causing the user to be a programmer. We use the term *DMM* to refer to a message-passing multiple instruction stream, multiple data stream (MIMD) computer where each processor has its own local memory and there is no shared memory. The terms decomposition and partition are used interchangeably to refer to the splitting up of data arrays into segments, each of which is distributed to a different processor; that processor is then said to own that segment, i.e. this data is stored in its local memory. A data distribution is a mapping of data to multiple processors in this way.

A data distribution may be static (the mapping of segments to processors is unchanged during program execution) or dynamic (the data-to-processor mapping changes at run-time, as decided either automatically by the parallelizing system or explicitly by the programmer). Dynamic distribution may be used to maintain a balanced computational load over the processors of a DMM during program execution. Where there is a conflict between the "lowest-cost" distributions (in terms of the amount of interprocess communication) of a given array at different points in a program, static distribution of that array in accordance with one of those "best" distributions would generally result in excessive interprocess communication at the other points in the program, since at each such point the best distribution is not in effect. Dynamic distribution enables the resolution of such conflicts, although it is important that the com-

munication incurred by the redistribution of an array (to resolve these conflicts and hence minimize communication during a computation) does not exceed the communication overhead which that redistribution was intended to reduce.

Some systems permit explicit interarray alignment. This is the explicit specification of a positional relationship between data structures; it may be defined in an indirect form, using an intermediate reference frame, or as a direct relationship between the data structures. For example, two 4×4 arrays A and B may be directly aligned such that their elements are overlapped as shown in Figure 1. When these arrays are subsequently distributed over processors their elements will be positioned in relation to one another as shown in Figure 1; for example, each shaded element of B is guaranteed to reside on the same processor as the shaded element of A aligned with it in the diagram.

Most of the systems discussed in this article produce target code in accordance with the single program multiple data (SPMD) model [3]. Under this scheme each processor runs the same program but executes different code depending on its processor id and the data held in its local memory, examining every statement to determine what part it must play, if any, in the execution of that statement.

In the owner-computes paradigm all computations updating a given datum are performed by the processor owning that datum. An alternative scheme is the owner-stores paradigm, whereby the right-hand side expression of an assignment is computed by a processor which owns data appearing in that expression and this result is then sent to the processor owning the left-hand side

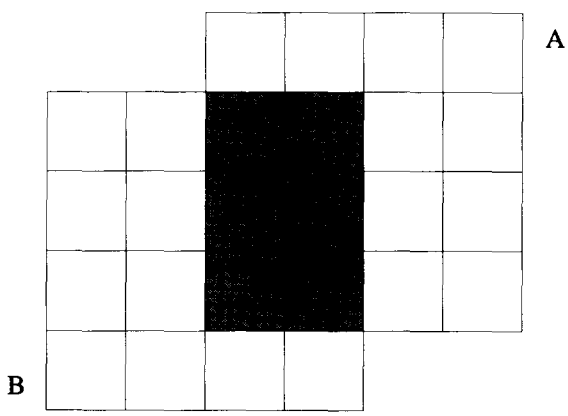

FIGURE 1 The alignment of two 4×4 arrays A and B.

```
      PROGRAM JACOBIRELAXATION
      REAL OLD, NEW
      DIMENSION OLD(128,128), NEW(128,128)
C     INPUT VALUES OF ARRAY 'OLD'
      ...
      DO 10 I = 2, 127
       DO 10 J = 2, 127
        NEW(I, J) = C * (OLD(I, J) + OLD(I-1, J) + OLD(I+1, J)
     &                 + OLD(I, J-1) + OLD(I, J+1))
10    CONTINUE
C     OUTPUT VALUES OF ARRAY 'NEW'
      ...
      END
```

FIGURE 2 Sequential algorithm for Jacobi relaxation on 128×128 grid.

datum; in some cases this scheme may incur less communication than the owner-computes paradigm.

The data-parallel programming style is a SIMD-like style, making use of a single execution thread and a global name space in expressing (loosely) synchronous operations. Regular computations are those for which all the necessary communications can be precisely determined at compile-time. Irregular computations, however, do not permit this—the data transfer behavior of the computation depends on the input with the result that communications can only be determined exactly at run-time. One example of irregularity is indirect array referencing of the form A[B[i]] where the array A is distributed. With a reference of the form B[i] the i is generally some loop counter whose range of values is known at compile-time so that the compiler can determine which communications statements must be generated for that subset of the iterations of the loop which is to be executed by a given processor (i.e., the set of other processors with which communication is necessary is determinable at compile-time). If, however, instead of i we have some expression that is completely indeterminable until execution time when the compiler cannot make any deductions regarding the communicants of a given processor; the subscript B[i] in the indirect reference A[B[i]] is an example. In this case if A is distributed (regardless of whether B is distributed) then we have an irregularity and suitable run-time facilities are required that the compiler can ensure are invoked during program execution. (Note that if A is not distributed, but is instead replicated, and B is distributed then there is no irregularity because the situation is simply equivalent to an ordinary occurrence of B[i].)

Figure 2 outlines a sequential algorithm, written in Fortran 77, for Jacobi relaxation on a grid of 128×128 points. The thrust of the algorithm is to

update each point in the grid using its north, south, east, and west neighbors, with special conditions at the boundaries. This is an example that requires the partitioning of data in a many-processor system. Where appropriate each of the following scheme descriptions includes an example of how this procedure could be implemented under that scheme. In each case the parallelization constructs are highlighted in bold type.

2.1 SUPERB

The SUPERB parallelization system [4–8] was completed in 1989 and was the first implemented system to transform FORTRAN 77 code (with accompanying data distribution description) into message-passing code for a DMM. It restructured sequential FORTRAN 77 code into SUPRENUM Fortran for execution on the SUPRENUM multiprocessor; message-passing Fortran for the Intel iPSC and GENESIS machines could also be generated. As each node in the SUPRENUM machine possessed a pipelined vector unit, parallelization consisted of two phases: MIMD parallelization (creating a set of processes) followed by vectorization (within each process). The SUPRENUM project was primarily aimed at the numerical simulation of large grid-based problems (typically having 10^6 to 10^9 grid points) where the computations at each grid point are mostly local.

SUPRENUM Fortran is an extended Fortran that includes the task concept (a task can be activated more than once, each activation creating a process) and Fortran 90-style array features. The SPMD and owner-computes models were observed and some compile-time optimizations, such as message vectorization and iteration elimination, were carried out. Irregular problems involving subscript indirection were supported; however, dynamic distribution and explicit interarray alignment were not. Scalar variables were replicated over all processors.

In the SUPERB system, the programmer interactively specifies data partitioning (by block) and distribution using a special notation (the original Fortran 77 code remains unaltered); the partitioning of an n-dimensional array is specified in the following form;

part *array-name* (sd_list_1, sd_list_2, . . . , sd_list_n)

Each sd_list_i is a list of segment descriptors specifying the segmentation of dimension i of the array; an sd_list_i may be a list of constant descriptors such as

$$L_1:R_1 - L_2:R_2 - \ldots - L_n:R_n$$

where Li and Ri are integer constants, or a list of variable descriptors such as

$$c_1*x_1 - c_2*x_2 - \ldots - c_n*x_n$$

where each integer constant c_i specifies a number of segments each of size x_i (integer constant or variable). The values x_i are determined by the system.

The following example illustrates the use of this notation in its simplest form where an array A is partitioned into four blocks in its second dimension and is left unpartitioned in its first dimension (note that the default lower bound Li in each case is 1):

```
part A (1, 4)
```

The above example makes use of a default (linear) processor arrangement. However, the target processor arrangement may be specified as a processor array structure (pas). For example, the following code declares GRID to be a two-dimensional abstraction of the underlying processors, whereas DIAG refers to those processors constituting the leading diagonal of GRID:

```
pas GRID (4, 4)
pas DIAG (4) with (i=1, 4 DIAG(i) → GRID(i, i))
```

This mechanism allows for considerable scope in the description of processor arrays because linear expressions are permitted in the processor-subset mapping.

As a further example consider the Fortran 77 code in Figure 2, assuming the GRID processor array structure, defined above, is used. To implement this by partitioning each of arrays OLD and NEW into contiguous segments, each of the size 32×32 elements and each allocated to one processor (assuming there are at least 16 processors), the user may specify the array decomposition using constant descriptors;

```
part OLD (1:32 - 33:64 - 65:96 - 97:128,
1:32 - 33:64 - 65:96 - 97:128)
part NEW (1:32 - 33:64 - 65:96 - 97:128,
1:32 - 33:64 - 65:96 - 97:128)
```

[or, because this example requires equal-sized blocks, the simpler form may be used

```
part OLD (4, 4)
part NEW (4, 4)]
```

or by using variable descriptors as in

```
part OLD (4*n, 4*n)
part NEW (4*n, 4*n)
```

The first use of constant descriptors above illustrates the possible specification of contiguous rectangular data segments of arbitrary size.

An array may be partitioned to only a subset of a given processor array structure; for example:

```
part B(4) with (i=1, 4 B(i) →
GRID(5−i, i))
```

maps the elements of B onto the secondary diagonal of GRID.

Alignment may be achieved using distribution variables. In the following, array C is distributed by block along DIAG (distribution variable j is defined [on its first appearance] to the width of these blocks); D is distributed likewise but with its first block of size (j + 11);

```
part C(4 <j>) with DIAG
part D(1 <j+11> − (3) <j> with DIAG
```

The user may further specify the parallelization process itself. Analysis services are provided by the system to enable the user to examine the communication overhead resulting from a chosen partition. The analysis phase provided by the system permits the inspection of the communication overhead resulting from a partition, after which the user can interactively change the partition specification and apply a choice of transformations to optimize communications; further optimizations may be chosen to improve vectorization.

Nonlocal read access to neighboring array data is provided by system-determined overlaps. These are private copies of adjoining nonlocal data; their consistency is maintained by interprocess communications generated by the SUPERB system. For the distribution specified above, applied to the Jacobi relaxation example (see Fig. 2), the system will ensure, by appropriate analysis of the references involved, a one-element-wide overlap around each block.

2.2 Id Nouveau

Rogers and Pingali [9, 10] present a compiler that transformed programs written in Id Nouveau into semantically equivalent C code for the iPSC/2. Id Nouveau is a functional language augmented with write-once arrays called I-structures. As in the case of imperative language arrays, the allocation of storage for an I-structure is separate from the definition to its elements; however, each element of an I-structure may only be defined once. I-structures therefore permit the incremental definition of arrays without the duplication overhead of functional language arrays. Id Nouveau also includes features for the specification of data domain decomposition.

Because the SPMD model of node program generation results in redundant activity (each node process examining every statement) the Id Nouveau compilation system applied compile-time resolution where possible. This is the specialization of the code of each node process to its local data. Greater run-time efficiency is achieved by virtue of the reduction of redundant activity and because, in general, this specialization makes the node programs different from one another the SPMD model is effectively abandoned. However, compile-time resolution cannot be applied in certain cases, such as irregular computations, where sufficient information is not available at compile-time. Run-time resolution must then be used as a last resort; although less efficient, this guarantees that such codes can be compiled. The Id Nouveau compiler could recognize opportunities for accumulation, a form of owner-stores strategy that entails the evaluation of the right-hand side of an assignment by the process most involved in providing the terms featured in the right-hand side expression; the owner-computes paradigm was otherwise applied as a default.

In the Id Nouveau compiler, data distribution is expressed within the source code using syntax extensions. For example, a scalar variable may be replicated to all processors using:

```
(variable_name : ALL)
```

or placed on a specified processor:

```
(variable_name : Pid)
```

where Pid uniquely identifies a particular processor. An array (I-structure) is distributed using one of three builtin, regular distributions; **blocks**,

wrapped **rows** (i.e., cyclically distributed), and wrapped **columns**. Figure 3, the Jacobi relaxation example, illustrates the use of the **block** distribution function in partitioning arrays OLD and NEW into contiguous blocks of size 32×32, to be distributed one block per processor.

```
Procedure Jacobi_relaxation (OLD: block(32, 32)) ; block(32, 32)
  { I-structures OLD and NEW are distributed block-wise in both dimensions}
  Let NEW = array (128 , 128) : block(32, 32) in
    for i = 2 to 127 do
      for j = 2 to 127 do
        NEW [i, j] = C * (OLD[i, j] + OLD[i-1, j] + OLD[i+1, j]
                          + OLD[i, j-1] + OLD[i, j+1]);
  return NEW
```

FIGURE 3 Id Nouveau code for Jacobi relaxation.

```
(* specify PxP processor array called Procrs *)
processors Procrs : array [1 .. P, 1 .. P] with P in 1 .. 4;

(* block decomposition of arrays OLD and NEW in each dimension, and *)
(* distribution of these blocks over Procrs *)
var OLD, NEW : array [1 .. 128, 1 .. 128] of real dist by [block , block] on Procrs

  (* input values of array OLD *)
  ...

  (* computational code *)
  forall i in 2 .. 127, j in 2 .. 127 on NEW[i, j].loc do
    NEW[i, j] := C * (OLD[i, j] + OLD[i-1, j] + OLD[i+1, j]
                      + OLD[i, j-1] + OLD[i, j+1]);
  end;

  (* output values of array NEW *)
  ...
```

FIGURE 4 Jacobi relaxation in Kali.

Array distributions are limited to the above three mappings and neither explicit interarray alignment nor dynamic distribution is supported. Consequently this system can support efficiently fewer applications than other languages such as Fortran D and Vienna Fortran (see later). However, array distribution specification is straightforward, requiring only the use of simple mappings, although knowledge of processor identification is required for the distribution of scalar variables.

2.3 Kali

Kali [11, 12] provides a set of parallelization extensions supporting sequential-style programming on distributed memory architectures. For development purposes Kali (which grew out of the BLAZE project by the same group) was implemented as a Pascal-based language, although it could be based on any other sequential language. The Kali compiler transformed a program written in this language into SPMD message-passing C code for the NCUBE/7 or iPSC/2. As far as possible the analysis required to produce the necessary communications and synchronizations was performed at compile-time; irregular problems were supported but these dictated that their analysis be done (less efficiently) at run-time, using inspector/executor loops. Kali did not support the explicit alignment of arrays or the dynamic distribution of data.

Figure 4 illustrates the use of Kali in implementing the Jacobi relaxation example. The programmer's first task is to specify an array of physical processors using a processors statement, in this example it is defined to be a two-dimensional P×P processor array called Procrs. The parameter P is chosen by the run-time system to be the largest possible integer constant in the given range (in this case 1..4).

Next the programmer must define how arrays are to be distributed over this target architecture. This is achieved by appending a distribution (**dist**) clause to the declarations of those arrays intended for distribution; scalar variables, and arrays declared without a distribution clause, are universally replicated. Within a distribution clause the programmer specifies the distribution pattern for each dimension of the data array, observing the limitation that the number of distributed dimensions in a distribution clause must equal the number of processor array dimensions. User-defined distribution patterns are possible but Kali additionally provides the intrinsics **block** and **cyclic**, illustrated below; block-cyclic distribution is also supported.

```
processors line : array 1 .. P]
with P in 1 .. 10;
var A : array [1 .. 100] of real dist
by [block] on line;
    B : array [1 .. 100] of real dist
by [cyclic] on line;
    C : array [1 .. 100, 1 .. 100] of
real dist by [*, block] on line;
    D : array [1 .. 100] of real;
```

Array A is distributed over the one-dimensional processor array "line" as contiguous blocks of 10 elements each, whereas the elements of B are distributed individually in a round-robin fashion. The asterisk indicates that a dimension is not to be distributed and so each processor in "line" will receive a block of 10 contiguous columns of C. Array D is undistributed and each processor in "line" receives a complete copy of D. In the ex-

ample of Figure 4 arrays OLD and NEW are distributed over Procrs as contiguous two-dimensional blocks.

Computations using distributed arrays must be enclosed in **forall** loops. These are treated as fully parallel loops and no provision is made for any parallelization of loops with interiteration dependences. Within a **forall** loop, the values used are those that were current immediately before the loop (a strategy referred to as "copy-in/copy-out semantics"). Furthermore, the programmer must append an **on** clause to **forall** loops, specifying which processor is to execute each iteration of the loop. Figure 4 illustrates the use of the **.loc** function for this purpose, which ensures that the iteration updating NEW[i, j] is executed on the processor owing NEW[i, j]. However, this need not be the case because it is possible to depart from the owner-computes paradigm by explicitly referencing processors in an **on** clause.

Kali presents the programmer with a relatively large set of parallelization concerns. In addition to specifying data distributions, the programmer must also declare the underlying processor topology and explicitly indicate not only parallel loops but also the processors on which the iterations of these loops are to be executed, i.e., the user must take responsibility for both data and iteration distributions.

2.4 ARF

Wu et al. [13] presented an experimental compiler and run-time support system, predominantly aimed at enabling the execution of sparse, unstructured applications written in ARF (ARguably Fortran), an extended dialect of Fortran 77. The ARF compiler produced an SPMD node program containing embedded PARTI primitives [14] to implement the necessary communications. PARTI (Parallel Automated Run-time Toolkit at ICASE) is a library of run-time procedures that support irregular distribution patterns and irregular computations involving subscript indirection. A run-time resolution scheme was used, employing an inspector/executor approach for communication preprocessing; even for regular computations, no message communications were firmly decided at compile-time.

Using ARF's language extensions, data distribution can be regular (**block** or **cyclic**) or user defined and irregular; the latter is achieved using a regularly distributed integer-valued mapping array of the same size and shape as the array to be distributed, as illustrated below:

```
distributed regular using block
real A(1000)
distributed regular using block
integer maparray(1000)
distributed irregular using
maparray real B(1000)
```

Here the processor to which B(i) is mapped is identified by the value of maparray(i). The current implementation of ARF can only support partitioning of one dimension (the last dimension) of an array, although the PARTI primitives are capable of supporting more general distributions. Neither dynamic data distribution nor explicit interarray alignment is supported.

The **distributed do** language extension indicates that the iterations of a DO loop are to be distributed over the processors of the target machine, whereas another extension, the **on** clause, gives the user a means of controlling this distribution. As a result, the owner-computes rule is not necessarily adhered to.

An example of the use of the ARF language in implementing the Jacobi relaxation problem is given in Figure 5. Note that only the last (i.e., the second) dimension of OLD and NEW can be partitioned and therefore these arrays are partitioned and distributed as blocks of columns, one block per processor. This example is tentative because the researchers state that the syntax accepted by the current version of the ARF compiler differs slightly from that presented by Wu et al. [13].

The ARF system provides relatively few parallelization extensions but in enabling the treatment of irregular distributions the system requires the programmer to have some knowledge of processor identification. The **on** clause and **distributed do** construct, although necessary for sufficient programmer control in certain kinds of application,

```
C     distribute contiguous blocks of columns of arrays OLD and NEW
      distributed regular using block real OLD(128, 128), NEW(128, 128)

C     initialisation of array OLD
      ...

      distributed do 10 j = 2 , 127
        do 10 i = 2, 127
          NEW(i, j) = C * (OLD(i, j) + OLD(i-1, j) + OLD(i+1, j)
     &                 + OLD(i, j-1) + OLD(i, j+1))
10    continue

C     output of array NEW
      ...
```

FIGURE 5 ARF code for Jacobi relaxation.

nevertheless increase the involvement of the programmer in parallelization.

2.5 ADAPT

Merlin [15, 16] presents a system called ADAPT (Array Distribution Automatic Parallelization Tool) that was developed under Esprit Project 2071 (PUMA). ADAPT transforms data-parallel programs written in distributed Fortran 90, a Fortran 90 subset enhanced with data-partitioning extensions, into a form suitable for execution on arrays of T9000 transputers with C104 switches (although the techniques are applicable to any message-passing MIMD system). ADAPT makes no attempt to parallelize DO loops; parallelism is obtained from the inherent parallelism of the Fortran 90 array features. There is therefore an onus on the programmer to maximize the use of such features.

ADAPT produces SPMD code in accordance with the owner-computes paradigm. This generated code takes the form of a Fortran 77 node program, including calls to communication procedures provided by a purpose-built communications library called ADLIB (Array Distribution LIBrary). The same node program is executed by each process in a multidimensional process array (because each transputer can support more than one process the researchers refer to *processes* rather than *processors*). The communication procedures of ADLIB are high-level grid-based routines requiring at least nearest-neighbor connectivity in every dimension of the process array. Indirect array referencing, expressible using (potentially distributed) vector subscripts, is supported. ADAPT is currently at an early stage of development and little emphasis has as yet been placed on optimizations.

The size of the logical process array is defined, in a separate file, in the form

```
proc_array = (D1, ... , Dn)
```

As an example, a two-dimensional 4×4 process array for use by the Jacobi relaxation code would be declared as follows:

```
proc_array = (4, 4)
```

Preparation of a Fortran 90 program for parallelization by ADAPT consists of the declaration of a **DISTRIBUTION** attribute for each array to be distributed. For an n-dimensional real array A this takes the form

```
REAL, DIMENSION (e1, ... , en),
DISTRIBUTION (d1, ... , dn) :: A
```

Each non-negative integer d_i indicates the contiguous block distribution of dimension i of A over the process array (block distribution is the only form of distribution available). A value of 0 for d_i indicates that dimension i of the data array is not to be distributed; a value $d_i > 0$ indicates that dimension i of the data array is distributed across dimension d_i of the process array. For example

```
REAL, DIMENSION (10, 10),
DISTRIBUTION (0, 1) :: A
```

results in the following distribution over a one-dimensional five-process array (Fig. 6).

Omission of a **DISTRIBUTION** attribute for an array causes that array to be undistributed. Such arrays are replicated to all processes in the process array, as are scalar variables. An array can be distributed over only a subset of the dimensions of the process array, in which case it is replicated over the remaining process-array dimensions. For example, with proc_array = (2, 4)

```
REAL, DIMENSION (8),
DISTRIBUTION (2) :: B
```

gives the distribution seen in Figure 7.

A dummy array argument may adopt its distribution from the corresponding actual argument, a

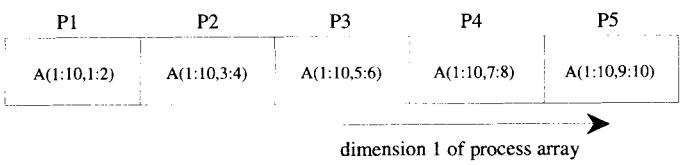

FIGURE 6 The distribution of array A.

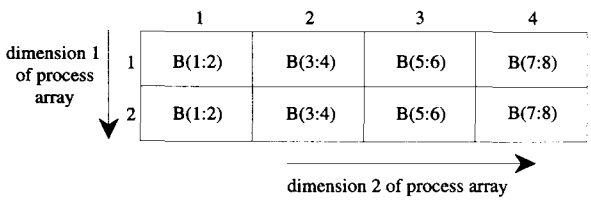

FIGURE 7 The distribution of array B.

strategy that the researchers call assumed distribution (Fig. 8). Explicit interarray alignment is not supported, nor is dynamic data distribution.

Apart from the definition of a process array in a separate file, the preparation of a distributed Fortran 90 program from its Fortran 90 equivalent entails the use of only a single, simple parallelization feature, **DISTRIBUTION**. However, the price paid for such simplicity is the relatively limited applicability of the current ADAPT system compared with other languages like Vienna Fortran and Fortran D. In fact this simplicity is deceptive because the programmer must also make effective use of the array features of Fortran 90 to maximize parallelism.

```
      PROGRAM JacobiRelaxation
!     assumes a 4x4 underlying array of processors, declared in another file
!     distribute dimensions 1 and 2 of arrays OLD and NEW over
!     dimensions 1 and 2 of the underlying processor array
      REAL, DIMENSION (128, 128), DISTRIBUTION (1, 2) :: OLD, NEW
!     input the values of array OLD
      ...
!     computational code
      NEW(2:127, 2:127) = C * (OLD(2:127, 2:127) + OLD(1:126, 2:127)
     &                         + OLD(3:128, 2:127) + OLD(2:127, 1:126)
     &                         + OLD(2:127, 3:128))
!     output values of array NEW
      ...
      END PROGRAM JacobiRelaxation
```

FIGURE 8 Jacobi relaxation in distributed Fortran 90.

2.6 Vienna Fortran

Some authors [17–19] describe Vienna Fortran, an extended dialect of Fortran 77 that provides the programmer with facilities for the specification of data distribution within conventional Fortran 77 code; there is also a Fortran 90 subset[20] with Vienna Fortran extensions. The Vienna Fortran compilation system, based largely on the achievements of the SUPERB project, is currently in an advanced stage of development. This system supports the full Fortran 77 language and targets the SUPRENUM, iPSC/860, and GENESIS machines; optimized message-passing code is generated in accordance with the SPMD paradigm. Vienna Fortran makes use of the PARTI primitives [14] to support the indirect referencing of distributed arrays.

The use of the Vienna Fortran extensions in the annotation of Fortran 77 code essentially comprises three main aspects:

1. The declaration of target processors.
2. The distribution of data arrays over the target processors.
3. The specification of parallel loops and the allocation of their iterations to processors.

Declaration

In any given Vienna Fortran program there is an implicitly declared one-dimensional array of target processors, called **$P**, which consists of all the processors available in the target machine. If any other processor structure is required then the programmer may superimpose that structure upon the SP arrangement, which is achieved using a **PROCESSORS** statement. For example

```
PROCESSORS procrs3D (N, N, N)
```

declares a three-dimensional array of processors, called procrs3D. The value of N in the above example is determined at load time in accordance with the number of processors available in the target machine. It is important to note that this processor array is merely an alternative view of the N×N×N target processors constituting **$P**; the indices of a given processor within **$P** and procrs3D are related according to the column-major ordering convention of Fortran 77. Individual processors may be referenced as elements in an array; for example, **$P**(2) is also procrs3D(2, 1, 1). Fortran 90 array section notation may also be used to reference subsets of processor structures, for example, procrs3D(1:4, 3, 9). An intrinsic function **$MYPROC** is provided which, when called by a node program executing on one of the processors, returns the processor's index within **$P**.

The processor structure declared in a **PROCESSORS** statement, such as procrs3D above, is known as the primary processors structure. If further alternative views of the processors of **$P** are required then these may be obtained by reshaping the primary processor structure, again in accordance with Fortran 77 column-major ordering. For example, if a two-dimensional structure were also needed then the above declaration might read

```
PROCESSORS procrs3D (N, N, N)
RESHAPE procrs2D (N, N×N)
```

The additional structures obtained by reshaping, such as procrs2D, are known as secondary processor structures. All processor arrays declared in a Vienna Fortran program must contain the same number of processors.

No particular interconnection between processors is assumed in either **$P** or any defined processor structures. For example, procrs2D is not necessarily connected as a nearest-neighbor grid.

Distribution

Some data arrays may not require distribution in a given application, for such arrays no Vienna Fortran annotations are required—the arrays are declared in the normal Fortran 77 manner and as a result are replicated on every processor. Scalar variables may also be universally replicated. However, in general some arrays will need to be distributed to achieve program speed-up through data-parallel execution. To this end Vienna Fortran provides an extensive and powerful set of features that enable the specification of a wide range of (static or dynamic) array distributions.

Static Distribution. A two-dimensional array A may be statically distributed over the processor structure procrs2D (i.e., in terms of this view of the target processors) by annotating its declaration in the following manner

```
REAL A(N, N×N)
DIST distribution-expression
TO procrs2D
```

The **TO** clause is optional; if it is omitted then the distribution occurs over the primary processor structure. The distribution-expression specifies a distribution type, which is a class of distributions described using distribution functions; a list of functions may be given, each of which defines the distribution pattern of one dimension of the array

11	12	13
21	22	23
31	32	33

FIGURE 9a Processor array p2D.

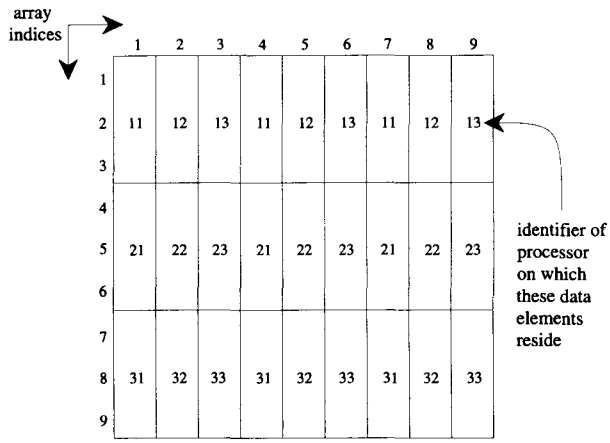

FIGURE 9b The distribution pattern of array B.

over a dimension of the target processor array, or a single distribution function may be given which defines the distribution pattern for the entire array. A range of intrinsic distribution functions are available that provide **BLOCK**, **CYCLIC** and block-cyclic (**CYCLIC**(*block-size*)) distributions of array dimensions. Examples are

```
PROCESSORS p2D (3, 3)
REAL B(9, 9) DIST (BLOCK, CYCLIC)
REAL C(90, 90) DIST (BLOCK, CYCLIC(10))
```

These distributions and the p2D grid are illustrated in Figure 9 (a, b, and c); processor id numbers are indicated in the boxes (for brevity processor (i, j) is indicated by ij). **BLOCK** distribution produces the distribution of an array dimension in equally sized contiguous sections; **CYCLIC** distribution produces a round-robin distribution of the individual elements along a dimension. In Figure

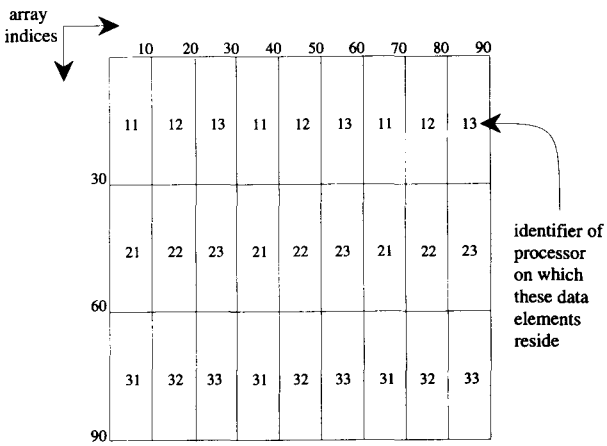

FIGURE 9c The distribution pattern of array C.

9c the second dimension of array C is partitioned into 10-element blocks that are placed cyclically onto processors.

The elision symbol ":" in place of a distribution function for a dimension of an array prevents the distribution of that dimension. For example, the distribution

```
REAL D(10, 100) DIST(CYCLIC, :) TO $P
```

cyclically distributes the rows of D over the one-dimensional processor array **$P** as shown in Figure 10 (assuming, for this example, that **$P** contains five processors).

In the case where the number of processor-array dimensions exceeds the number of data-array dimensions being distributed the entire array is replicated over the extra dimensions of the processor array.

Programmers may define their own distribution functions, for example

```
   DFUNCTION distfunc
   TARGET T(1:)
   DO 10 I = 1, SIZE(T)
    T(I) DIST TO $P(SIZE(T) - I)
10 CONTINUE
   END DFUNCTION distfunc
```

The **TARGET** array T in the definition of distfunc represents the array being distributed. This simple distribution function may be used to specify the distribution of an array F. For example

```
REAL F(10) DIST (distfunc)
```

achieves a reverse-order distribution of the elements of F over the processors of **$P** (assuming a sufficient number of processors).

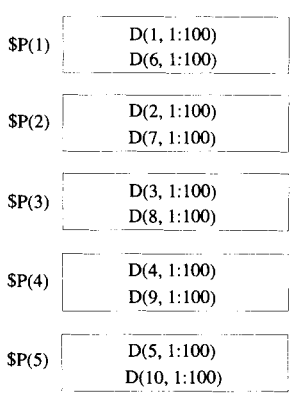

FIGURE 10 Cyclical distribution of array D over **$P**.

```
      PROGRAM JACOBIRELAXATION
      PROCESSORS grid2D(4, 4)
      REAL OLD(128, 128) DIST (BLOCK, BLOCK)
      REAL NEW(128, 128) DIST (=OLD)
C     INPUT VALUES OF ARRAY 'OLD'
      ...
      DO 10 I = 2, 127
       DO 10 J = 2, 127
        NEW(I, J) = C * (OLD(I, J) + OLD(I-1, J) + OLD(I+1, J)
     &                 + OLD(I, J-1) + OLD(I, J+1))
10    CONTINUE
C     OUTPUT VALUES OF ARRAY 'NEW'
      ...
      END
```

FIGURE 11 Vienna Fortran code for Jacobi relaxation.

The distribution of an array may alternatively be specified using the distribution functions constituting the distribution-expression of another array. For example,

```
REAL G(2000, 20, 300)
DIST (CYCLIC, CYCLIC, BLOCK)
REAL H(100, 2500)
DIST (=G.3, =G.1) TO procrs2D
```

distributes the first dimension of **H** by **BLOCK** (the distribution function of G.3, the third dimension of G) and the second dimension of H in **CYCLIC** fashion (in accordance with the distribution of G.1, the first dimension of G). This feature is further illustrated in the Jacobi relaxation example given in Figure 11. This code declares a two-dimensional array of 16 processors, called grid2D, and distributes the array OLD over grid2D in contiguous blocks of size 32×32 elements. The array New is distributed in the same way by virtue of the (=OLD) distribution expression. Note that although Vienna Fortran makes no assumption concerning the interconnection patterns of target processors, clearly the annotations in Figure 11 will minimize the communications overhead in the case of the target processors being connected in a nearest-neighbor manner.

The foregoing distributions are all examples of the direct specification of distributions. Vienna Fortran also allows the implicit distribution of one array (called the target array) in terms of the distribution of another array (the source array), i.e., interarray alignment. This is achieved using the **ALIGN** keyword, for example:

```
REAL K(100, 100) ALIGN K(I, J)
WITH H(J, I*10)
```

aligns each element of the target array K with the source array element identified by evaluating the subscript expressions of the source array H. I and J are placeholders, i.e., bound variables in this annotation that each range from 1 to 100 (their corresponding subscript ranges in array K). Hence, for example, target element K(5, 21) is aligned with source element H(21, 50).

Programmers can also define their own alignment functions, for example:

```
AFUNCTION alfunc
TARGET T(i:)
SOURCE S(1:)
DO 10 I = 1, SIZE(T)
 T(I) ALIGN WITH S((I+6)
          MODSIZE(S)+1)
10 CONTINUE
END AFUNCTION alfunc
```

This alignment function may be used to specify the alignment of an array L to a four-element array M thus

```
REAL L(10) ALIGN (alfunc) WITH M
```

which results in the following alignment of elements:

```
M(1) ↔ L(2), L(6), L(10)
M(2) ↔ L(3), L(7)
M(3) ↔ L(4), L(8)
M(4) ↔ L(1), L(5), L(9)
```

It is possible to define irregular data distributions in Vienna Fortran where individual elements of an array may each be mapped to a specified processor using the **INDIRECT** distribution function and an integer-valued mapping array of the same shape and size as the data array. This mapping array may itself be distributed.

```
INTEGER map (10) DIST(CYCLIC)
REAL Q(10) DYNAMIC
...
DISTRIBUTE Q :: INDIRECT (map)
```

In the above example the value of map(i) is the index within **$P** of the processor to which Q(i) is to be mapped.

Dynamic Distribution. Vienna Fortran also provides for the dynamic distribution of arrays. Such an array is distinguished by an additional annotation to its declaration, the **DYNAMIC** keyword. Examples are:

```
REAL R(100) DYNAMIC, DIST(CYCLIC) TO $P
REAL U(100) DYNAMIC
```

The array R is initially distributed cyclically but this distribution can later be altered, by virtue of its **DYNAMIC** declaration. The array U has no initial distribution and must not be accessed until it has been distributed. The distributions that a dynamically distributed array is permitted to adopt at run-time can be limited by specifying explicitly the allowed distributions. For example

```
REAL V(100) DYNAMIC,
RANGE(BLOCK, CYCLIC)
```

specifies that V may only be distributed in a block or cyclic fashion. Any other distribution of V will have an undefined effect.

The alignment and initial distribution of dynamic arrays are specified in the same way as for static arrays; the array to which a dynamic array is aligned may be either static or dynamic. Such alignment is not maintained if either array is later redistributed. Such an association can, however, be achieved using the **CONNECT** keyword. For example

```
REAL W(100, 100) DYNAMIC,
DIST (CYCLIC, BLOCK) TO procrs2D
REAL X(100, 100) DYNAMIC, CONNECT (=W)
```

Here W is called the primary array and X is a secondary array. A primary array and the secondary arrays **CONNECT**ed to it constitute a connect set. A dynamic array may be a member of only one connect set. Only the primary array in a connect set may be redistributed and when this happens each of its secondary arrays is redistributed in a manner related to the primary's new distribution by that secondary's **CONNECT**ion. The **CONNECT**ion in the above example specifies that the distribution type of X will always be that of W.

Dynamic distribution is specified by a **DISTRIBUTE** statement of the form

```
DISTRIBUTE A1, ... , An :: distrib
[NOTRANSFER (Aj, ... , Ak)]
```

On execution of this statement each listed dynamically distributed array A_i is given the distribution distrib, which may be a direct, **INDIRECT**, or im-

plicit specification as described above. For any primary array distributed by the **DISTRIBUTE** statement its secondary arrays are also distributed in accordance with their **CONNECT**ions. The optional **NOTRANSFER** clause attributes new access functions to the listed arrays $A_j, \ldots, A_k$ (which are selected from the list $A_1, \ldots, A_n$ and their **CONNECT** sets) in accordance with the specified distribution distrib but does not produce any transfer of their data; the previous data values of the arrays $A_j, \ldots, A_k$ are subsequently ignored.

It must be noted that although dynamic distribution directives are provided, it is the user's responsibility to ensure that they are used wisely, especially that their use does not incur greater redistribution costs than the costs (of suboptimal execution with unredistributed arrays) that the redistribution is intended to alleviate. This decision may be far from trivial; tools are needed to help the programmer in making such decisions. Another part of the VFCS is a static performance estimation module [21] that may be of some use in this respect.

Other Distribution-Related Features: Control Constructs. Vienna Fortran provides two features, the **IF** construct and the **DCASE** construct, that enable the distribution of an array to dictate the flow of execution. For example

```
REAL Y(1000, 100) DYNAMIC
...
IF (IDT(Y, (BLOCK, BLOCK))) THEN
  if-code
ENDIF
```

the *if-code* will only be executed if both dimensions of Y are block distributed; **IDT** (Identical Distribution Types) is an intrinsic inquiry function that compares the distribution of an array with a specified distribution type.

In the following example of a **DCASE** construct the code to be executed is determined by the first pair (in textual order) of **CASE** limb distribution expressions to match the actual distributions of AA and BB; the asterisk signifies "any distribution."

```
REAL AA(1000, 100), BB(200, 200)
DYNAMIC ...
SELECT DCASE (AA, BB)
  CASE (BLOCK, CYCLIC), (BLOCK, BLOCK)
    code1
  CASE (CYCLIC, *), (BLOCK, CYCLIC)
    code2
  CASE (DEFAULT)
    coden
END SELECT
```

Other Distribution-Related Features: Subroutine Parameters. The distribution of a formal parameter in a subroutine can be static or dynamic. For each formal parameter a distribution is specified which is enforced at subroutine entry. If the formal parameter is dynamic, however, then its distribution may be inherited from the actual argument by specifying the annotation **DIST**(*). A **RANGE** clause may also be used to specify the permissible distributions of a dummy argument with inherited distribution, thereby providing the compiler with useful information that may not otherwise be determinable. For example

```
REAL Z(N) DIST(*) RANGE((CYCLIC(10)),
(BLOCK))
```

declares that the formal parameter Z inherits its distribution from the corresponding actual parameter and that this distribution will either be a block-cyclic pattern with block size 10 or a simple block distribution. If the actual argument is statically distributed then any redistribution performed within the subroutine is undone at exit. Such distribution restoration may optionally be enforced, using the **RESTORE** keyword, for dynamically distributed actual arguments. A **NOTRANSFER** attribute can be given to specify that any redistribution carried out on entry to a subroutine involves only a change in access function and no movement of data. A local array can be aligned with a formal parameter or given its own distribution. Where appropriate actual arguments may be specified using Fortran 90 array section notation.

Parallel Loops

Vienna Fortran provides a **FORALL** loop construct that enables the programmer to assert that the iterations of a loop may be executed in parallel by virtue of their being independent (i.e., the data written within one iteration are neither read nor written within any other iteration of the loop).

Loop iterations may be assigned to specified processors, for example

```
FORALL I = 1, N ON $P(PROC(I))
...
END FORALL
FORALL I = 1, M ON OWNER(V(I))
...
END FORALL
```

it is assumed that PROC is some array, defined elsewhere, whose contents may be used as processor indices. **OWNER** is a Vienna Fortran intrinsic function that identifies the home processor of its argument. In the default case, when the **ON** clause is omitted, the loop iterations are assigned by the compiler. This may be carried out so as to minimize communication, perhaps splitting individual iterations across several processors, or a simple (inefficient) assignment of several iterations to a single processor may be enforced.

FORALL loops are implicitly synchronized at start and finish. They may be (tightly) nested and may contain private variables, in which case each iteration is equipped with its own copy of those variables. Reduction statements, using intrinsic and user-defined reduction functions, may be used within the loop and their results become available at the end of the loop. Vienna Fortran also provides I/O support for concurrent file access by individual processors to several storage devices.

Summary

Vienna Fortran provides the programmer with a comprehensive range of features that enable the efficient parallelization of a wide range of algorithms coded within the conventional Fortran 77 programming paradigm and referencing a single (virtual) shared memory space. Although Vienna Fortran provides the expressive control needed to specify the parallelization of even quite pathological algorithms, it has in so doing significantly increased the complexity of the programmer's task and consequently increased the possibility of (potentially very elusive) errors.

Nevertheless, this increased involvement of the programmer in the parallelization process is much more palatable than the disadvantages of message-passing programming and clearly may be justified by the program execution speed-ups achievable. Indeed the programmer requiring a simple one-dimensional processor array and only static distributions need only specify the appropriate data distributions.

2.7 Fortran D

A few authors [22–25] describe an extended Fortran, called Fortran D, that enables a programmer to specify the distribution of data and computational work across a DMM. Currently a Fortran 77D (i.e., extended Fortran 77) compiler is being developed at Rice University and Wu and Fox [26] are developing a Fortran 90D (extended Fortran 90) compiler at Syracuse University. Ultimately these two projects will converge with a single definition of Fortran D, the current "official" version of which is summarized here. It is proposed that the Fortran D compiler will form part of a data-parallel programming system that will also include a static performance estimator (to provide the user with predictions of relative performances of a Fortran D program with different data distribution [27]) and an automatic data partitioner (which will make use of the static performance estimator either by interactively assisting a user in finding an efficient data distribution or by automatically producing one). The Fortran D compiler will produce optimized code in the SPMD model.

The annotation of Fortran code using the Fortran D extensions essentially comprises four main components:

1. The optimal specification of the number of target processors.
2. The mapping of data arrays onto intermediate frames of reference (called decompositions).
3. The distribution of decompositions over the target processors (implying the distribution of the arrays mapped onto these decompositions).
4. The specification of parallel loops and the allocation of their iterations to processors.

This categorization shows some similarity to that given in the previous section for Vienna Fortran. The significant difference is the use of an intermediate mapping device (the decomposition) in Fortran D, which is intended to promote code portability.

Specification

The required number of processors may be stipulated at the begining of a Fortran D program using the reserved variable **n$proc**; alternatively this may be omitted and the number of processors will be determined automatically at run-time according to availability.

Mapping

Data distribution begins with the specification of one or more decompositions. A decomposition does not occupy any storage; it is simply an abstract structure that can be regarded as a frame of reference for interarray alignment and as a vehicle for the distribution of arrays. An array intended for distribution is first aligned with a decomposition using placeholders (I, J, K, etc.) as in the following examples. Arrays for which no alignments are specified are replicated over all processors.

1. REAL A(N, N), B(N, N)
   DECOMPOSITION DEC1(N, N)
   ALIGN A(I, J) **with** DEC1(I, J)
   ALIGN B(I, J) **with** DEC1(I, J)

   Here a two-dimensional N×N decomposition called DEC1 is declared and arrays A and B aligned to it; on the distribution of DEC1, A and B will be codistributed. The above two ALIGNments can alternatively be stated as

   ALIGN A, B with DEC1
2. REAL C(N, N), D(N, N)
   DECOMPOSITION DEC2(N, N)
   ALIGN C(I, J) **with** DEC2(6*I, J)
   ALIGN D(I, J) **with** DEC2(I, 3*J-2)

   Here D has a stride of 3 and an offset of -2 in the J dimension; C(1, 4), for example, will be codistributed with D(6, 2), as shown in Figure 12 (assuming N=8).
3. REAL E(N, N)
   DECOMPOSITION DEC3(N, N)
   ALIGN E(I, J) **with** DEC3(J, I)

   This is an example of permutation, in this case the transpose of the array E is aligned with the decomposition.
4. REAL F(N, M, N)
   DECOMPOSITION DEC4(N, N)
   ALIGN F(I, J, K) **with** DEC4(I, K)

   Here the second dimension of F is undistributed so that elements in its J dimension are collapsed together in the eventual distribution. This is illustrated in Figure 13 for the N=4 case.
5. REAL G(N, N)
   DECOMPOSITION DEC5(N, N, N)
   ALIGN G(I, J) **with** DEC5(I, J+1, 3)

   This is an example of embedding, the mapping of an array onto a decomposition that has more dimensions. Depending on the distribution of its decomposition it might be the case that such an array is not mapped over all the processors in the target machine.

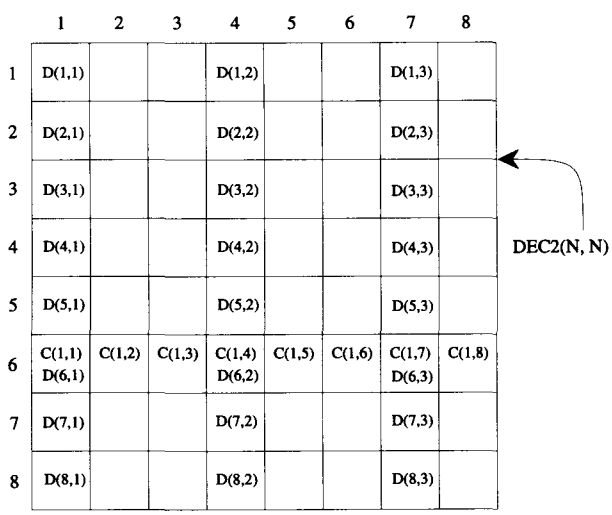

FIGURE 12 The alignment of arrays C and D with the decomposition DEC2.

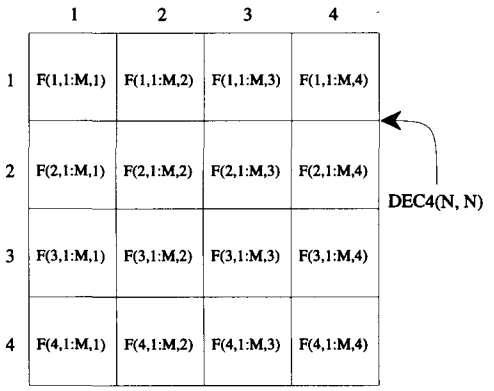

FIGURE 13 Mapping of array F onto decomposition DEC4, showing the collapsing of the J dimension of F.

It is possible to specify the mapping of an array onto a decomposition in such a way that some of its elements are mapped onto nonexistent positions in the decomposition. Fortran D therefore provides for an **ALIGN** statement with an optional **overflow** clause that specifies one of three options

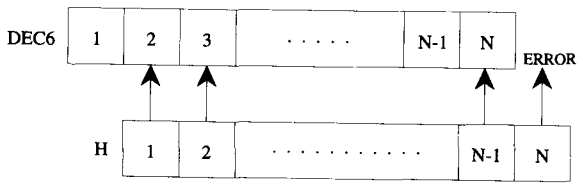

FIGURE 14a The alignment of array H with decomposition DEC6.

(**ERROR, TRUNC,** and **WRAP**) per dimension. This is used to describe how array elements extending beyond the decomposition are to be treated, for example

```
REAL H(N), K(N, N)
DECOMPOSITION DEC6(N), DEC7(N, N)
ALIGN H(I) with DEC6(I+1) overflow (ERROR)
ALIGN K(I, J) with DEC7(I-1, J+1)
overflow (TRUNC, WRAP)
```

In this example, the element H(N) is aligned with DEC6(N+1). This alignment is specified with type **ERROR** (the default type when the **overflow** clause is omitted); this means that H(N) is unmapped and attempts to access it are illegal (see Figure 14a). The **TRUNC** option causes the overflowing elements (here the first row of K) to be mapped to the overflowed edge of the decomposition; hence the first and second rows of K are both mapped to the first row of DEC7. The WRAP option maps the overflowing elements to the opposite end of the decomposition; the last column of K is mapped to the first column of DEC7. These alignments are illustrated in Figure 14b.

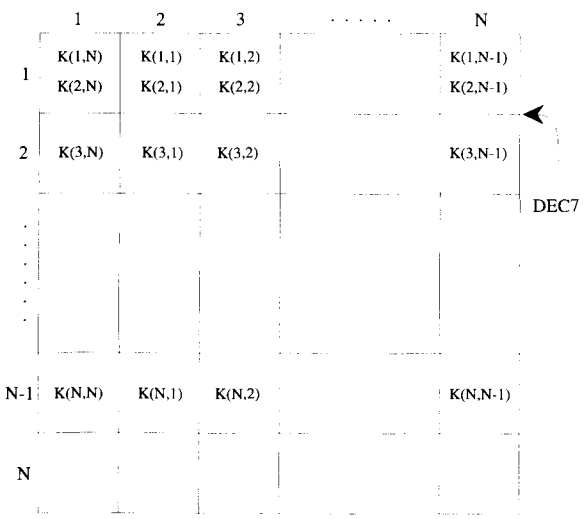

FIGURE 14b The alignment of array K with decomposition DEC7.

The foregoing **ALIGN** statements mapped entire arrays onto decompositions. However, it is also possible to map only part of an array where, for example, a large work array is to be subdivided into a collection of smaller logical arrays at runtime. This partial mapping is achieved by specifying a section of the array in a **range** clause. The following example illustrates that all rows of L (indicated by the asterisk), but only columns 1 to N, are to be mapped.

```
REAL L(N, N+N)
DECOMPOSITION DEC8(N, N)
ALIGN L(I, J) with DEC8(I, J)
range(*, 1:N)
```

The replication of array elements over a dimension of a decomposition is specified by the programmer indicating a range of a decomposition dimension rather than a placeholder, for example

```
REAL M(N), P(N), Q(N, N)
DECOMPOSITION DEC9(N, N)
ALIGN M(I) with DEC9(I, 2:5)
ALIGN P(I) with DEC9(*, I+5)
ALIGN Q(I, J) with DEC9(J, *)
```

This example is illustrated in Figure 15 where each of the second, third, fourth, and fifth columns of DEC9 is associated with the whole of M. For every row of DEC9 there is an association between its last (N-5) elements and the first (N-5) elements of P. Each column of Q is mapped to every element in the corresponding row of DEC9.

Distribution

The distribution of an array over the target machine is achieved by specifying its associated decomposition in a **DISTRIBUTE** statement; the execution of such a statement distributes the arrays **ALIGN**ed to the specified decomposition. The syntax for the n-dimensional case is

DISTRIBUTE *decomposition* (*attribute*$_1$, . . . , *attribute*$_n$)

Each attribute specifies the manner in which that dimension of the decomposition is to be distributed over the target machine. The attribute * indicates no distribution and as a result the corresponding dimension is allocated locally. Three

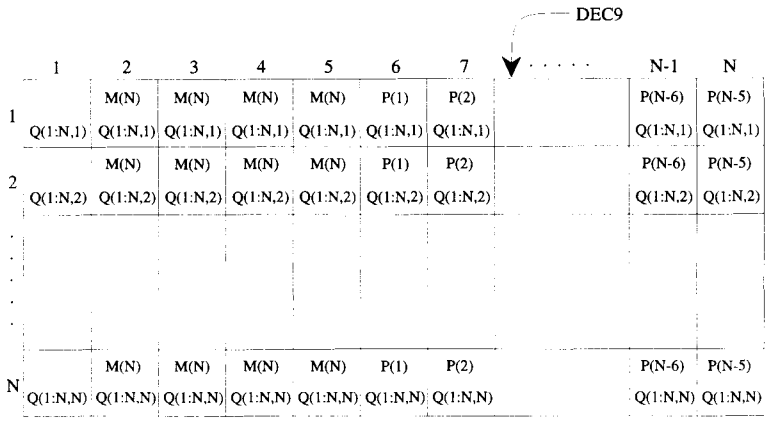

FIGURE 15 The alignment of arrays M, P, and Q with decomposition DEC9.

regular distribution attributes are available, namely **BLOCK**, **CYCLIC**, and **BLOCK_CYCLIC**; their use implicitly creates a processor array in that the target processors are allocated as evenly as possible between the dimensions.

```
DISTRIBUTE DEC9 (BLOCK, *)
DISTRIBUTE DEC10 (CYCLIC,
BLOCK_CYCLIC (2) )
```

The above examples are illustrated in Figure 16 for the case where **n$proc = 4**. Figure 16a shows the first dimension of the decomposition DEC9 partitioned into contiguous blocks distributed between the processors p1 to p4; the remaining dimension of DEC9 is not distributed. The elements of DEC10 (assumed to have been declared with size 8×8) are distributed individually, in a round-robin fashion, in one dimension and grouped into blocks of size 2 in the other dimension, these blocks also being distributed cyclically as shown in Figure 16b.

Another example of the use of the **DISTRIBUTE** statement is given in Figure 17 for the Jacobi relaxation example where 16 target processors are specified and a decomposition DD of size 128×128 is declared. Having mapped arrays OLD and NEW directly onto DD it is then distributed in **BLOCK** fashion in both dimensions over the target processors. Because processors are allocated evenly between the dimensions of a decomposition this example causes DD (and hence the arrays OLD and NEW) to be partitioned and distributed as 16 (i.e., 4×4) contiguous blocks, each of size 32×32 elements.

Extended forms of the regular distribution attributes are provided, allowing the programmer to specify explicitly the number of processors allocated to each dimension. As virtual processors are not supported in Fortran D the programmer must ensure that the specified attributes do not require more processors than **n$proc**. The number of

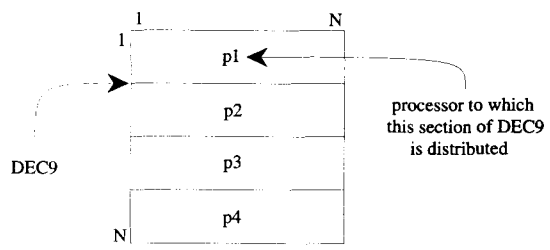

FIGURE 16a The distribution pattern of the N×N decomposition DEC9.

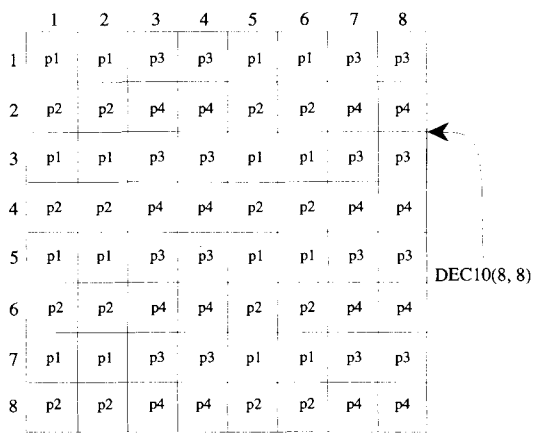

FIGURE 16b The distribution pattern of the 8×8 decomposition DEC10.

```
      PROGRAM JACOBIRELAXATION
      n$proc = 16
C     16 TARGET PROCESSORS DECLARED
      REAL OLD, NEW
      DIMENSION OLD(128, 128), NEW(128, 128)
      DECOMPOSITION DD(128, 128)
      ALIGN OLD, NEW with DD
      DISTRIBUTE DD(BLOCK, BLOCK)
C     BOTH OLD AND NEW ARE NOW PARTITIONED IN A
C     4X4 FASHION (AS BLOCKS OF SIZE 32X32) AND
C     DISTRIBUTED OVER 16 TARGET PROCESSORS.
C     INPUT VALUES OF ARRAY 'OLD'
      ...
      DO 10 I = 2, 127
        DO 10 J = 2, 127
          NEW(I, J) = C * (OLD(I, J) + OLD(I-1, J) + OLD(I+1, J)
     &                    + OLD(I, J-1) + OLD(I, J+1))
10    CONTINUE
C     OUTPUT VALUES OF ARRAY 'NEW'
      ...
      END
```

FIGURE 17 Fortran D code for Jacobi relaxation.

processors per dimension is specified as an extra parameter. Taking the distribution of the decomposition DD in the Jacobi relaxation example, if rather than allocating the 16 target processors evenly between its two dimensions, giving the 4×4 scheme shown in Figure 18a, we had instead required the 2×8 scheme illustrated in Figure 18b then the **DISTRIBUTE** statement would have been written as follows

```
DISTRIBUTE DD(BLOCK(2), BLOCK(8))
```

Irregular distributions are achieved in Fortran D by using replicated or distributed mapping arrays of integers in a manner analogous to the use of the **INDIRECT** distribution function in Vienna Fortran. In the following example element R(I, J) is distributed to the processor identified by the value of the mapping array element MAP(I, J)

```
  n$proc = 16
  REAL R(4, 4)
  INTEGER MAP(4, 4)
  DECOMPOSITION DEC11(4, 4),
DEC12(4, 4)
  ALIGN R with DEC11
  ALIGN MAP with DEC12
  DISTRIBUTE DEC12(CYCLIC, BLOCK)
C FILL MAP WITH PROCESSOR ID NUMBERS
  . . .
  DISTRIBUTE DEC11(MAP)
```

Fortran D also supports the dynamic alignment and distribution of arrays where both **ALIGN** and **DISTRIBUTE** are executable statements. As the example below illustrates, however, Fortran D differs from Vienna Fortran by not discriminating between statically and dynamically distributed arrays.

```
  REAL S(N, N), T(N, N)
  DECOMPOSITION DEC13(N, N)
  ALIGN S, T with DEC13
  DISTRIBUTE DEC13(CYCLIC, CYCLIC)
C BOTH S AND T ARE DISTRIBUTED
(CYCLIC, CYCLIC)
  DISTRIBUTE DEC13(BLOCK, BLOCK)
C BOTH S AND T ARE NOW REDISTRIBUTED
(BLOCK, BLOCK)
  ALIGN T(I, J) with DEC13(J, I)
C THE TRANSPOSE OF T IS NOW ALIGNED
WITH S
```

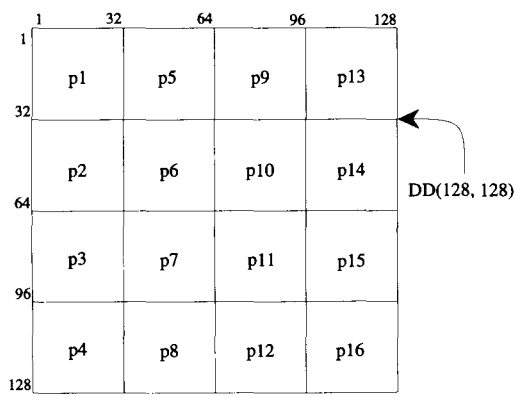

FIGURE 18a The mapping of decomposition DD onto 16 processors, resulting from the even allocation of processors between dimensions.

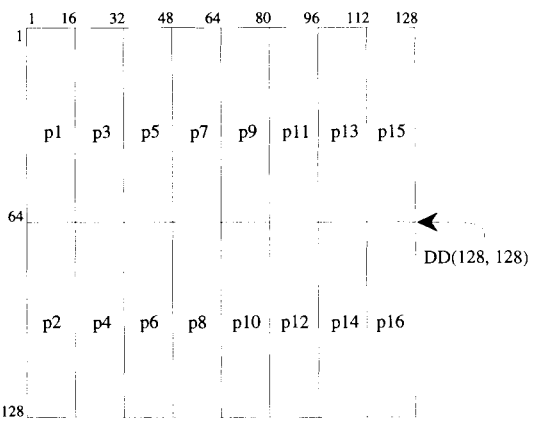

FIGURE 18b The mapping of decomposition DD onto 16 processors, resulting from an uneven allocation of processors between dimensions.

Distributed arrays may be used as actual parameters to procedures and such an array may be dynamically redistributed within a procedure. However, unlike the Vienna Fortran equivalent such redistribution cannot be maintained outside the procedure because Fortran D limits the effect of a **DECOMPOSITION, ALIGN,** or **DISTRIBUTE** to the scope of the enclosing procedure. Another difference with Vienna Fortran is the lack of a facility for the querying of distribution patterns at run-time.

Although ordinary sequential-style DO loops may be used for regular computations, situations can arise where a compiler cannot fully exploit the inherent parallelism in such a loop (e.g., irregular computations) and must make worst-case assumptions about interiteration dependences. In these cases, if the programmer knows that parallel execution will be possible then, as in Vienna Fortran, a **FORALL** loop may be specified instead; communication-free, determinate parallel execution of the loop iterations is then obtained (although communication may still be required before and after the loop for nonlocal values). In each iteration of a **FORALL** loop only values defined before the loop or within that iteration may be used. **FORALL** loops may be nested.

Although the code produced by the compiler is by default based on the owner-computes model, it is possible to override this using an **ON** clause, which specifies on which processor each iteration of a **FORALL** loop will execute, for example

```
n$proc = 16
REAL U(400, 400), V(400, 400),
W(400, 400), X (400, 400)
DECOMPOSITION DEC14(400, 400)
ALIGN U, V, W, X with DEC14
DISTRIBUTE DEC14(BLOCK, BLOCK)
. . .
FORALL I = 201, 400
 FORALL J = 1, 400 ON HOME (X(I-200, J))
  X(I, J) = (U(I-200, J) + V(I-200, J)
+ W(I-200, J)) * X(I-200, J)
 ENDFOR
ENDFOR
```

In this example it will probably be more efficient to execute each assignment on the processor owning the right-hand side values rather than on that holding X(I, J), thereby implementing the owner-stores paradigm. This is achieved by the use of the **HOME** function, which returns the identifier of the processor owning the specified datum X(I-200, J) and is analogous to the **OWNER** intrinsic function in Vienna Fortran.

As with Vienna Fortran, the range of applications that may be efficiently parallelized using Fortran D is extensive, but the comprehensive set of extended features that it provides makes possible a substantial increase in the involvement of the programmer in the program parallelization process and a corresponding increase in the complexity (and error proneness) of that task.

2.8 Booster

Paalvast et al. [28, 29] describe the Booster language, a subproject of the ParTool Parallel Processing Environment project. Booster enables the description of parallel algorithms, based on array-like data structures, for **both** shared memory multiprocessors and DMMs. Booster introduces the concepts of index and data domains. An index domain consists of ordered index sets (each of which is a finite set of tuples of integers) and a data domain consists of data values of certain types. Different syntaxes are used for manipulations on index and data domains; data manipulations are imperative whereas index manipulations are functional.

The only data structure provided is the shape, which is a finite set of elements whose values are all of a single data type; each element is uniquely associated with an index of the shape's index set. Shapes differ from conventional arrays in that a shape-index set may be more complex than the simple linear indexing of an array. Selected shape elements are referenced using views. A view is not a data structure but is constructed from the index sets of one or more shapes. Effectively the view is an abstraction of array-like access and removes the need for index loops.

Examples of these concepts can be seen in Figure 19 which is the algorithm module for an implementation of the Jacobi relaxation method. A Booster program consists of a collection of separately compiled modules of which there are two types, an algorithm module and an annotation module (considered later). in this example, OLD and NEW are declared as shapes of size 128 × 128 elements and the computation shows the use of the simple view [1:126, 1:126] applied to the shape NEW and other simple views applied to OLD to effect the update.

In the next example the shape S is declared as a rectangular 3 × 4 data structure and V is a view identifier defined as a view on S such that V[0, 0],

```
MODULE Jacobi (OLD) -> NEW
SHAPE OLD (128#128) OF REAL;
      NEW (128#128) OF REAL;
BEGIN
  NEW[1:126, 1:126] := C * (OLD[0:125, 1:126] + OLD[1:126, 1:126] +
                            OLD[2:127, 1:126] + OLD[1:126, 0:125] +
                            OLD[1:126, 2:127]);
END.
```

FIGURE 19 Algorithm module for an implementation of Jacobi relaxation in Booster.

V[0, 1], . . . , V[2, 3] reference S[0, 0], S[0, 1], . . . , S[2, 3], respectively.

```
SHAPE S(3#4) OF REAL;
V ← S;
```

This view identifier V may then be redefined or used to define other views of S, for example

```
V1 ← V[0:2, 3]
```

defines the view identifier V1 so that it references the fourth column of shape S.

Irregular computations may be expressed in Booster using content selection views as follows

```
SHAPE A(10) OF REAL;
      B(10) OF INT;
. . .
. . . A[B<4] . . .
```

The view [B<4] is a content selection view because the Boolean expression B<4 results in an index set whose elements reference the values of B which obey this expression; this index set is then applied to A. Hence if B is the set {2, 1, 6, 6, 3, 7} then the index set B<4 is {1, 2, 5} and the elements referenced in A are A[1], A[2], and A[5]. Clearly irregularity will result when A is distributed because the precise elements of A that are being referenced cannot be determined until run-time.

The algorithm modules of a Booster program are machine independent and as a result information regarding the decomposition and distribution of data over processor memories, and the assignment of computation responsibility to processors, must be provided by the programmer in an annotation module using an annotation language.

Within an annotation module, the programmer first specifies a virtual machine that serves as a model onto which data and associated computation responsibility may be mapped. The processor structure of the virtual machine may be defined separately from its memory structure, for example

```
VIRTUAL MACHINE sharedmem
(PROC procr(p), MEM memory(m));
```

declares a machine called sharedmem with a single memory of size m, shared among p processors, whilst

```
VIRTUAL MACHINE distribmen
(PROC procr(p), MEM memory(n) (m));
```

declares a machine called distribmem consisting of p processors and n memory units each of size m.

In the annotation module Jacobi (Figure 20) the virtual machine VM consists of a processor-plus-memory arrangement PM made up of 16 identical processors, each with its own local memory of size 2×32×32. The module also defines the mapping of the shape OLD and of the associated responsibility for the computation of its elements, onto the virtual machine VM. In the statement

```
OLD [i, j] ← VM [(i div 32)*4
+ j div 32, 0, i mod 32, j mod 32];
```

The first subscript in VM[...] defines the processor responsible for performing assignments to the element OLD[i, j]. A variant of the owner-computes convention is employed—the processor responsible for assignment to a shape element on the left-hand side of an assignment statement is also responsible for the calculation of the expression on the right-hand side. The remaining subscripts in VM[...] define the location on the local memory of the processor identified by the first subscript, into which the element OLD[i, j] is to be mapped. Interarray alignment is practised using the virtual machine as a reference frame.

Thus the shape OLD is partitioned into 16 (i.e., 4×4) contiguous blocks, each of which is stored in the local memory of one of the 16 processors of VM. The same distribution is performed on the shape NEW, giving the mapping shown in Figure

```
ANNOTATION MODULE Jacobi;
VIRTUAL MACHINE VM (PROCMEM PM (16)(2#32#32));
IMPORT Jacobi :: OLD, NEW (128#128);
OLD [i, j] <- VM [(i div 32)*4 + j div 32 , 0 , i mod 32 , j mod 32];
NEW[i, j] <- VM [(i div 32)*4 + j div 32 , 1 , i mod 32 , j mod 32];
REAL MACHINE RM (PROCMEM RPM (16)(lmax#mmax#nmax));
VM[i, l, m, n] <- RM[i, l, m, n];
END.
```

FIGURE 20 Annotation module for an implementation of Jacobi relaxation in Booster.

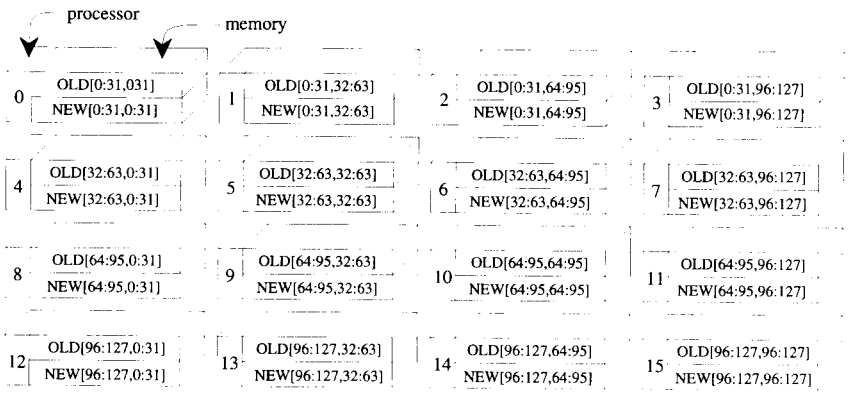

FIGURE 21 The mapping of shapes OLD and NEW over the virtual machine VM.

21. Each local memory must therefore be capable of storing two 32×32 blocks of data.

The programmer may also define a real machine and a mapping of the virtual machine. The processors of the real machine need not be identical and, unlike the virtual machine, a real machine possesses an interconnection structure, although this is only visible to the compiler and not to the programmer. An example of a real machine is the machine RM in Figure 20. This consists of a processor-plus-memory arrangement RPM which comprises 16 processor-plus-memory units, each memory being of size lmax X mmax X nmax (where lmax ≥ l, mmax ≥ m, and nmax ≥ n). These processors might be arranged as a 4×4 grid with a nearest-neighbor interconnection structure. Figure 20 specifies a very simple mapping from VM to RM, with each virtual processor being mapped onto its own real processor and each virtual machine memory location being mapped onto its real counterpart.

Although the above example defines a mapping in terms of a shape identifier (OLD and NEW), giving a static distribution, it is possible to define a mapping in terms of a view identifier instead. Unlike a shape, the size of a view may change at runtime; consequently a mapping defined in terms of a view identifier may also change, thereby achieving dynamic distribution. For example, in Figure 22a VW is declared as a view on shape SH. In each WHILE iteration the view VW is redefined such that it shrinks: initially the correspondence between view VW and shape SH is

```
VW[0] ≡ SH[0], VW[1] ≡ SH[1],
VW[2] ≡ SH[2], VW[3] ≡ SH[3]
```

but after one iteration VW is redefined so that the correspondence becomes

```
VW[0] ≡ SH[0], VW[1] ≡ SH[1]
```

(in Figure 22a, lwb and upb are lower bound and upper bound, respectively).
The corresponding annotation module is given in Figure 22b which introduces a virtual machine and defines a mapping.

Initially the value of the parameter VWsize is 4, giving the mapping shown in Figure 23a. However, after one iteration VWsize has the value 2 and the mapping is as shown in Figure 23b. Dynamic distribution has occurred because shape element SH(1) has been moved from the local memory of processor 0 to that of processor 1, thereby achieving load balancing for the next phase of the computation.

```
MODULE shrinker (SH) -> (SH)
SHAPE SH(4) OF REAL;
BEGIN
  VW <- SH;
  WHILE SIZE (VW) > 0 DO
    computation
    VW <- VW[lwb:upb-2];
  END;
END.
```

FIGURE 22a Algorithm module for "shrinking view" example.

```
ANNOTATION MODULE shrinker;
VIRTUAL MACHINE virt (PROCMEM procr(2)(2));
IMPORT shrinker :: VW(VWsize);
VW[i] <- virt[i div (VWsize div 2), i mod (VWsize div 2)];
END.
```

FIGURE 22b Annotation module for "shrinking view" example.

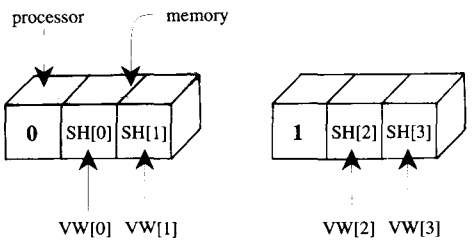

FIGURE 23a Initial state of virtual machine virt; the relationship between view VW and shape SH is also indicated.

An accompanying calculus, called V-cal, has been developed as a formal basis for Booster. The algorithm modules constituting the computational parts of a program are translated into an equivalent V-cal representation of the program. Transformations and optimizations are performed on this V-cal representation. The information contained in the annotation module is then translated into V-cal form; this is integrated with the V-cal representation of the computational code and the result undergoes some further optimizations. Finally an equivalent parallel program is generated using the SPMD model.

The implementation of a compiler to translate Booster programs to Fortran and C is currently in progress. The Booster parallel software development strategy is experimental and iterative with the compiler returning feedback information to the programmer which will, for example, enable an estimation of the amount of parallelism lost or introduced by different mappings, and the detection of communication hot-spots. Booster provides constructs that enable the specification of alternative mappings. The choice between alternatives is made by the compiler, so these constructs do not imply any dynamic distribution ability (i.e., they are not executable). The choice construct is of the (self-explanatory) form:

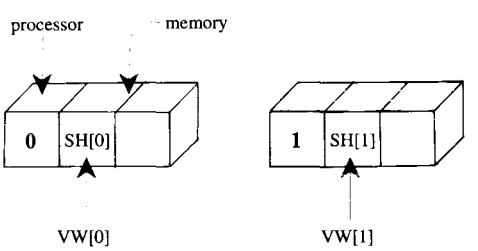

FIGURE 23b State of virtual machine virt after one iteration; the relationship between view VW and shape SH is also indicated.

IF *condition* **THEN** *mapping-statements*$_1$
ELSE *mapping-statements*$_2$

where the *condition* may be dependent on, for example, the size of a shape; the *alternative* construct is of the form:

ALTERNATIVE *mapping-statements*$_1$
*{**OR** mapping-statements*$_i$*}* ***END***

This construct specifies a list of mapping strategies, one of which is chosen by the compiler. The compiler will also inform the programmer of the annotations that it has chosen in the case of **ALTERNATIVE** annotations or in cases where mappings have not been provided by the programmer. An example of the latter is an assignment statement in which some of the participant shapes have no mappings defined. In such a situation the compiler may select mapping annotations such that (relevant dimensions of) these shapes are mapped in the same way as an already mapped participant shape. The compiler may also make use of data dependence information in such situations or simply select a predefined built-in mapping. The compiler feedback information allows the programmer to improve upon chosen annotations and perhaps also the computational code.

The Booster system differs significantly from the other systems outlined in that its source language is not based on an existing, well-known language. Booster contains several novel concepts that present a considerably greater barrier to the new user than the simple, relatively intuitive language extensions employed by the other systems. Furthermore, it is perhaps unfortunate that even the simplest mappings (such as block distribution) must be defined explicitly—no intrinsic mappings are available to the programmer—although this same feature allows the specification of relatively irregular mappings. The separation of mapping information (annotations) from the algorithm enables experimentation with different mappings, and even different machines, without altering the computational code. See summary table (Table 1).

3 DATA PARTITIONING AND DISTRIBUTION IN OTHER SYSTEMS

This section outlines some of the other systems that have made contributions towards the devel-

Table 1. Summary Table

System	Source Language	SPMD	Owner-Computes	Alternative to Owner-Computes	Explicit Interarray Alignment	Irregular Problems	Dynamic Distribution	Intrinsic Distribution Functions	Irregular or User-Defined Distributions	Distribution Query	Parallel Loops
SUPERB	Fortran 77	Yes	Yes	No	No	Yes	No	Yes	work space	No	No
Id Nouveau	Id Nouveau	No	Yes	Yes	No	Yes	No	Yes	No	No	No
Kali	Pascal	Yes	Yes	Yes	No	Yes	No	Yes	Yes	No	Yes
ARF	Fortran 77	Yes	Yes	Yes	No	Yes	No	Yes	Yes	No	Yes
ADAPT	Fortran 90	Yes	Yes	No	No	Yes	No	Yes	No	No	No
Vienna Fortran	Fortran 77 and 90	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Fortran D	Fortran 77 and 90	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Booster	Booster	Yes	Variant		Yes	Yes	Yes	No	Yes	No	N/A

opment of language constructs for data partitioning and distribution.

The source language for Pandore II [30, 31] is a subset of C, with data distribution syntax extensions from which message-passing DMM code is generated. The SPMD and owner-computes paradigms are employed; however, irregular computations are not supported. A Pandore II source program is a sequential program called distributed phases. A distributed phase is similar to a procedure in that its definition is given a name and a formal parameter list and its body is sequential code (the source language for Pandore II does not contain any parallel constructs). A distributed phase may only be called from within the main program. The partitioning (into blocks) and distribution of the data arrays used in a distributed phase are specified in the formal parameter list of the phase (called its distributed parameter list). After the partitioning of an array into blocks, the blocks are distributed over the processors of the target machine. The mapping of the blocks onto processors can be specified using one of two mapping styles, **regular** (contiguous allocation of blocks to processors) or **wrapped** (cyclic allocation). No particular processor arrangement is assumed in Pandore II; the user may specify the number of processors at compile-time using the command line interface. It is assumed that there is an efficient routing system in the target machine.

ADAPTOR (Automatic DAta Parallelism TranslatOR) [32] is a source-to-source translation package that translates programs written in a subset of Fortran 77 (extended with some CM Fortran features and many of the array-syntax features of Fortran 90) into message-passing Fortran 77 host and node programs for the iPSC/860 hypercube; other targets include the Meiko Concerto and the Parsytec GCel. The user consults an interactive transformation tool, XAdaptor, which provides analysis information on user-selected code units that the user can use to alter the source code and to insert data distribution directives; these directives may be used to specify block or cyclic distribution of the last one or two dimensions of an array. The generated code incorporates calls to message-passing communication routines from a DALIB (Distributed Array LIBrary). ADAPTOR does not support dynamic redistribution.

DINO (DIstributed Numerically Oriented language) [33] was one of the first systems in this area to be implemented (1986). It comprises standard C extended with high-level constructs for the description of parallel numerical algorithms for DMMs. There are three key concepts in DINO: environments, distributed data, and composite procedures. An environment consists of data and procedures and is equivalent to a process; the user, in declaring an environment structure, effectively defines a virtual parallel machine to fit the communications and number of processes required by a parallel algorithm. Data structures are distributed over this virtual machine/environment structure by specifying one-to-one or one-to-many mappings that may be user defined (and

hence potentially irregular) or selected from a set of built-in functions offering block, cyclic, and replicate distributions. All data distributions are static and explicit alignment is not supported. A composite procedure is a set of identical procedures, one in each environment in a given structure, that are called concurrently. DINO requires not only explicit parallel programming (in the form of composite procedures) but also the explicit marking of nonlocal accesses, using the '#' symbol.

Dataparallel C[34] is a SIMD-extended C variant and derivative of the C* language [35]. The programmer must specify groups ("domains") of virtual processors and the local computations and data for these domains. A global name space is supported but nonlocal references must be prefixed by a reference to the appropriate domain instance (the virtual processor owning the data). Predefined and user-defined static data mappings are possible. Dataparallel C compilers exist for shared memory multiprocessors (Sequent Symmetry S81) and DMMs (iPSC/2, nCUBE 3200).

Koelbel and others [36, 37] describe a compiler that accepts programs written in BLAZE (a largely sequential language but with functional procedure calls) and annotated with array distribution details. The compiler automatically generates equivalent E-BLAZE code where E-BLAZE is a superset of BLAZE, which effectively provides a virtual target architecture for the compiler. Parallel loops are specified using a **forall** construct. Data distributions are static and there is no provision for explicit alignment of arrays. The BLAZE project has been targeted at nonuniform memory access (NUMA) machines, such as the BBN Butterfly and the IBM RP3; its successor, Kali, targets DMMs.

Baber's Hypertasking system [38] translates C code annotated with data (block) distribution directives into message-passing code for the iPSC; other directives enable the delineation of loops that iterate over local data only. Distributed arrays are prohibited from being passed in procedure calls but dynamic redistribution is provided.

Carriero and others [39, 40] present the Linda parallel programming model. This is a memory model, based on the idea of tuple space and making use of the Linda coordination language in orchestrating coarse-grain parallel processes, which have been programmed in, for example, C code. Distributed data structures are used to provide a shared memory abstraction and can be regarded conceptually as free-floating, delocalized structures that are accessible simultaneously by several processes.

Crystal [41, 42] is a high-level functional language compiled for execution on a DMM by a compiler capable of implementing automatic data decomposition. Consequently, no indication of data partitioning/distribution need be supplied by the programmer. On compilation a Crystal program is divided into different computational phases, each represented by an index domain; each phase has associated with it a set of data fields that are interrelated by data dependence. Data arrays are heuristically aligned with index domains and a variety of block distributions are supported. Crystal has also been used as an intermediate language in the Crystallizing Fortran project, transforming Fortran programs for execution on massively parallel machines.

Another compiler capable of automatic data decomposition is ASPAR [43] for C or Fortran 77 programs. ASPAR recognizes four general types of loop and uses pattern-matching techniques to detect common reference patterns, or stencils, in the program. Using a knowledge base, a given stencil and loop type direct the selection of collective communication calls in the message-passing target program and an array within the loop is statically distributed as contiguous blocks of elements. A major drawback is that ASPAR makes some assumptions that can result in the semantic modification of the program.

Paragon [44] is a programming environment supporting the execution of SIMD programs on DMMs. Data distribution is performed by either the user or the system; user-specified, arbitrary, contiguous, rectangular data distributions are permitted, although only the first two dimensions of a given array may be distributed. Array redistribution is supported but explicit alignment is not.

The AL language [45] is compiled for the WARP distributed memory systolic array. Distributed arrays are specified as such in **DARRAY** declarations. Only one dimension of such an array may be distributed and given the programmer's indication of this dimension the AL compiler automatically generates a distribution.

The MIMDizer [46] is a commercially available programming environment targeting both shared and distributed memory MIMD machines. For DMMs the user interactively selects block, cyclic, or replicate distributions (maintained in a separate file) for a chosen array dimension; the user is also interactively involved in introducing parallel-

ism by specifying code spreading of loops, hence, like SUPERB, this system is not fully automatic after the data distribution has been specified.

Ruhl and Annaratone [47] present the ETHZ Oxygen compiler for the K2 experimental distributed memory machine. This system differs from the others in that it uses a functional rather than a data-driven parallelization strategy. The user inserts directives in the Fortran source code to indicate task-level and loop-level parallelism, reductions, and broadcast communications. Arrays may be private, replicated, or distributed (in a row-oriented, column-oriented, or ring fashion).

4 CONCLUSIONS

Features such as dynamic data distribution, irregular data distributions, support for irregular computation, circumvention of the owner-computes rule, interarray alignment (and the ability to maintain such an association after redistributions), run-time querying of distribution patterns, etc. are all desirable for ensuring the efficient parallel execution of a wide range of applications on DMMs. The systems presented in Section 2 of this article vary widely in the range of such features made available to the user and in the depth to which the user may become involved in the parallelization process.

In the ideal case a parallelizer would take responsibility for all aspects of parallelization because users of such systems will generally be non-computer scientists who wish to be involved as little as possible in the parallelization process, while seeking the maximum possible performance of their applications. However, certain aspects of distributed memory parallelization are intractable for even the most capable system; not least of these is the NP-completeness of the "shapes" problem (that of finding an optimal storage pattern for parallel execution in the general case) as proved by Mace [48].

Hence, for the majority of problems, user assistance is required and because data distribution has a significant effect on the performance of a parallelized program a sufficiently wide and expressive range of features must be provided by a parallelizing compiler to enable the specification of sufficiently precise distribution specifications for a wide range of problem types. The greater the control afforded by the provision of these features the greater the penalty incurred, namely, the erosion of the shared memory abstraction. A balance must therefore be determined between, on the one hand, taking the responsibility for parallelization away from the users and, on the other, providing them with the control needed to obtain efficient parallel code. In other words automation and high performance are, in general, mutually exclusive.

In general, the user cannot avoid giving at least some thought to the formulation of data parallelization annotations. Although these annotations will insulate the user from the real technicalities of DMM programming (processes, message-passing communication, and so on), this abstraction will be destroyed if appropriate debugging facilities are not provided; otherwise the user will be faced with the formidable task of debugging message-passing target code which, even if the user is familiar with the message-passing paradigm, will not have been seen previously.

Finally it must be pointed out that these parallelizing compilers complement but do not replace the programming of DMMs by explicit message-passing techniques. The situation is analogous to the use of high-level languages to write uniprocessor code, where assembly language may be used for the most performance-critical cases. DMM programming systems such as those suggested by this article may be used for the ease of use and reduction of development time whereas lower-level message-passing methods may be used in cases where performance is particularly critical and none of the available parallelizing systems can provide the required facilities.

ACKNOWLEDGMENTS

Considerable thanks must go to Barabara Chapman of the University of Vienna for information regarding the SUPERB system and for her helpful comments. Philip Crooks is supported by a Research Studentship (Distinction Award) from the Department of Education for Northern Ireland.

REFERENCES

[1] D. B. Loveman (ed.), "*High Performance Fortran: Language Specification*," *Sci. Programming*, vol. 2, pp. 1–167, 1993.

[2] D. B. Loveman, "High performance Fortran," *IEEE Parallel Distrib. Technol.*, pp. 25–42, February 1993.

[3] A. H. Karp, "Programming for parallelism," *IEEE Computer*, pp. 43–57, May 1987.

[4] H. P. Zima, H.-J. Bast, and M. Gerndt, "SU-

PERB: A tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, vol. 6, pp. 1–18, 1988.

[5] M. Gerndt, "Automatic parallelization for distributed-memory multiprocessing systems," Ph.D. thesis, Institut für Informatik, Bonn University, December 18, 1989.

[6] H. P. Zima, H.-J. Bast, M. Gerndt, and P. J. Hoppen, "SUPERB-The SUPRENUM parallelizer, Bonn," Research Report 861203, Institut für Informatik III, Bonn University, December 1986.

[7] H. Zima and B. Chapman, "Compiling for distributed-memory systems", Austrian Center for Parallel Computation, Technical Report ACPC/TR 92-17, November 1992.

[8] B. Chapman, e-mail dialogue with P. Crooks regarding SUPERB data partitioning notation. May 1993.

[9] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference", in *Proceedings of the ACM-SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, OR, June 1989*. pp. 69–80.

[10] K. Pingali and A. Rogers, "Compiling for Locality", in *Proceedings of the 1990 International Conference on Parallel Processing, St. Charles, IL, June 1990*. pp. II142–II146.

[11] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting shared data structures on distributed memory architectures," Technical Report ASD-TR 915, Department of Computer Science, Purdue University, Department January 1990.

[12] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Trans. Parallel Distrib. Systems*, vol. 2, pp. 440–451, 1991.

[13] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani, "Distributed memory compiler design for sparse problems," NASA Contractor Report 187515, ICASE Report No. 91–13, January 1991.

[14] H. Berryman and J. Saltz, "A manual for PARTI runtime primitives," Interim Report 90-11, ICASE, 1990.

[15] J. Merlin, "ADAPTing Fortran 90 Array Programs for Distributed Memory Architectures", in *Proceedings of the First International Conference of the Austrian Centre for Parallel Computation, Saltzburg, September 1991.*

[16] J. Merlin, "Techniques for the automatic parallelization of distributed Fortran 90," Esprit Project 2701 (PUMA) Deliverable Report 4.3.3, University of Southampton, November 1991.

[17] B. M. Chapman, P. Mehrotra, and H. P. Zima, "Vienna Fortran-A Fortran Language Extension for Distributed Memory Multiprocessors," in *Compilers and Runtime Software for Scalable Multiprocessors*, J. Saltz & P. Mehrotra, Eds. Amsterdam: Elsevier, 1991.

[18] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," Sci. Programming, vol. 1, pp. 31–50, 1992.

[19] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran-a language specification: Version 1.1," ICASE Interim Report 21, March 1991.

[20] S. Benkner, B. Chapman, and H. Zima, "Vienna Fortran 90, presented at Scalable High Performance Computing Conference 1992, Williamsburg, VA."

[21] T. Fahringer, R. Blasko, and H. Zima, "Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems," in *Proceedings of ACM International Conference on Supercomputing 1992, Washington, DC.*

[22] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng, "An overview of the Fortran D programming system," Centre for Research on Parallel Computation CRPC-TR91121, Department of Computer Science, Rice University, March 1991.

[23] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, "Fortran D Language Specification," April 15, 1991. Presented at High Performance Fortran Forum, Houston, TX, January 27-28, 1992.

[24] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiler support for machine-independent parallel programming in fortran D," Rice COMP TR91-149, Department of Computer Science, Rice University, January 1991.

[25] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiler optimizations for Fortran D on MIMD distributed-memory machines," Rice COMP TR91-156, Department of Computer Science, Rice University, April 1991.

[26] M-Y. Wu and G. Fox, "Compiling Fortran 90 programs for distributed memory MIMD parallel computers," SCCS-88; CRPC-TR91126, Syracuse Center for Computational Science, Syracuse University, April 1991.

[27] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator of guide data partitioning decisions," SCCS-14, Syracuse Center for Computational Science, Syracuse University, November 1990.

[28] E. M. R. M. Paalvast, "Programming for parallelism and compiling for efficiency," Ph.D. thesis, TNO Institute of Applied Computer Science, Delft, The Netherlands, January 1992.

[29] E. M. Paalvast, H. J. Sips, and L. C. Breebaart, "Booster: A high-level language for portable parallel algorithms," *Applied Numerical Math.*, vol. 8, pp. 177–192, 1991.

[30] F. André, O. Chéron, and J-L. Pazat, "Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II," in *Proceedings*

of the 3rd Workshop on Compilers for Parallel Computers, July 1992, Vienna. pp. 231–242.

[31] F. André, J-L. Pazat, and H. Thomas, "Pandore-A system to manage data distribution," INRIA Prog. 2, Project No. 1195, March 1990.

[32] T. Brandes, "Efficient data parallel programming without explicit message passing for distributed memory multiprocessors," Internal Report (draft) AHR-92-4, High Performmance Computing Center, German National Research Institute for Computer Science (GMD).

[33] M. Rosing and R. B. Schnabel, "An Overview of DINO-A New Language for Numerical Computation on Distributed Memory Multiprocessors," in *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing 1987.* pp. 312–316.

[34] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones, "Data-parallel programming on MIMD computers," *IEEE Trans. Parallel Distribut. Systems*, vol. 3, pp. 377–383, 1991.

[35] J. R. Rose and G. L. Steele, "C*: An extended C language for data parallel programming," Technical Report PL 87-5, Thinking Machines Corp., Cambridge, MA, 1987.

[36] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Semi-Automatic Domain Decomposition in BLAZE," in *Proceedings of the 1987 International Conference on Parallel Processing (IEEE).* pp. 521–524.

[37] P. Mehrotra and J. Van Rosendale, "The BLAZE language: A parallel language for scientific programming," *Parallel Comput.*, vol. 5, pp. 339–361, 1987.

[38] M. Baber, "Hypertasking Support for Dynamically Redistributable and Resizable Arrays on the iPSC," in *Proceedings of the 5th Distributed Memory Computing Conference, 1990.* pp. 59–66.

[39] N. Carriero, D. Gelernter, and J. Leichter, "Distributed data structures in Linda," in *13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, 1986.* pp. 236–242.

[40] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[41] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Systems*, vol. 2, pp. 361–376, 1991.

[42] J. Li and M. Chen, "Index domain alignment: Minimizing cost of cross-referencing between distributed arrays," in *Proceedings of Frontiers 90: The 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 1990.* pp. 424–433.

[43] K. Ikudome, G. Fox, A. Kolawa, and J. Flower, "An automatic and symbolic parallelization system for distributed memory parallel computers," in *Proceedings of the 5th Distributed Memory Computing Conference, Charleston, SC, April 1990.* pp. 1105–1114.

[44] A. Reeves, "The Paragon programming paradigm and distributed memory compilers," Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NI, June 1990.

[45] P. S. Tseng, "A parallelizing compiler for distributed memory parallel computers," in *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June 1990.*

[46] Pacific Sierra Research Corporation, "MIMDizer user's guide, version 7.02," Technical Report, 1991.

[47] R. Ruhl and M. Annaratone, "Parallelization of Fortran code on distributed-memory parallel processors," in *Proceedings of the 1990 ACM International Conference on Supercomputing, Amsterdam, The Netherlands, June 1990.*

[48] M. Mace, *Memory Storage Patterns in Parallel Processing.* Boston, MA: Academic, 1987.