HIGH PERFORMANCE FORTRAN SUPPORT FOR THE PARADIGM COMPILER

BY

EUGENE WILSON HODGES IV

B.S., North Carolina State University, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ABSTRACT

Lacking a global address space, distributed-memory multicomputers present a difficult programming model. High Performance Fortran (HPF) provides a portable, efficient environment for the programmer writing data-parallel applications. Research in the PARADIGM compiler project seeks to alleviate the programmer's burden of writing explicitly parallel applications using non-portable libraries and language extensions by automatically compiling sequential Fortran programs into efficient SPMD message-passing programs. Because the technology to automatically partition data and detect all parallelism in a sequential program is not mature, HPF provides a standard interface for supplying additional information to the compiler. The information expressed in an HPF program is complementary to the PARADIGM compiler project. HPF allows the programmer to assist the compiler with the process of efficiently compiling code that may eventually be automated, as well as providing information that is impossible to determine at compile time or efficiently at run-time. This thesis presents an overview of HPF, the implementation of HPF in the PARADIGM compiler, and an evaluation of the compiler using several benchmarks. The results demonstrate the effectiveness of the compiler in generating efficient SPMD Fortran code from the HPF code.

DEDICATION

To my wife, Mallory

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for supervising this work. Antonio Lain and Amber Roy-Chowdhury deserve thanks for helping to test and debug the HPF compiler pass. Ernesto Su helped generate some of the SPMD Fortran message-passing code with a new version of his compiler pass. Finally, I must thank Dan Palermo for answering my questions about PARADIGM.

I am indebted to my wife, Mallory, for her love, encouragement, and help. Finally, I wish to thank my family for all of their encouragement, support, and patience.

This work is supported by an Office of Naval Research Graduate Fellowship, for which I am very grateful.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Parallel computing provides scientists and engineers the capability to explore larger and more intricate problems that were previously unmanageable. However, the challenge of parallel programming has slowed the acceptance of parallel computing. To continue to expand the usage of parallel computers, compilers must assume more of the burden of parallelizing programs. The goal of the PARADIGM compiler project is to lessen the burden of parallelization by compiling sequential Fortran programs into efficient parallel code. This thesis furthers that goal by adding high-level directives for the programmer to use to assist the compiler.

1.1 Overview of PARADIGM

Distributed-memory multicomputers, such as the Intel iPSC/860, the Intel Paragon, the IBM SP-2, the NCUBE/2, and the Thinking Machines CM-5, offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. However, lacking a global address space, distributed-memory multicomputers present a very difficult programming model in which the programmer must write code to partition the data and computations across processors and to determine which sections of data need to be communicated among which processors during execution.

The PARADIGM compiler project [1]-[5] seeks to provide an automated means of parallelizing programs, written in a serial programming model, for execution on distributed-memory multicomputers. The compiler accepts sequential programs written in Fortran 77 and generates a Single Program, Multiple Data (SPMD) parallel program, including all necessary message-passing code.

Figure 1.1 shows a diagram of PARADIGM. In addition to performing traditional compiler optimizations, PARADIGM addresses many other issues within a unified framework: automatic data distribution [6], communication optimizations [4], support for irregular computations [7], exploitation of functional and data parallelism [8], and multithreaded execution [9]. PARADIGM uses *Parafrase-2* [10] to transform the sequential program into an intermediate representation and to analyze the code to generate flow, dependence, and call graphs.

1.2 High Performance Fortran

Because the technology to automatically partition data and detect all parallelism in a serial program is not mature, High Performance Fortran (HPF) [11] provides a standard interface for supplying additional information to the compiler. HPF is an extension to

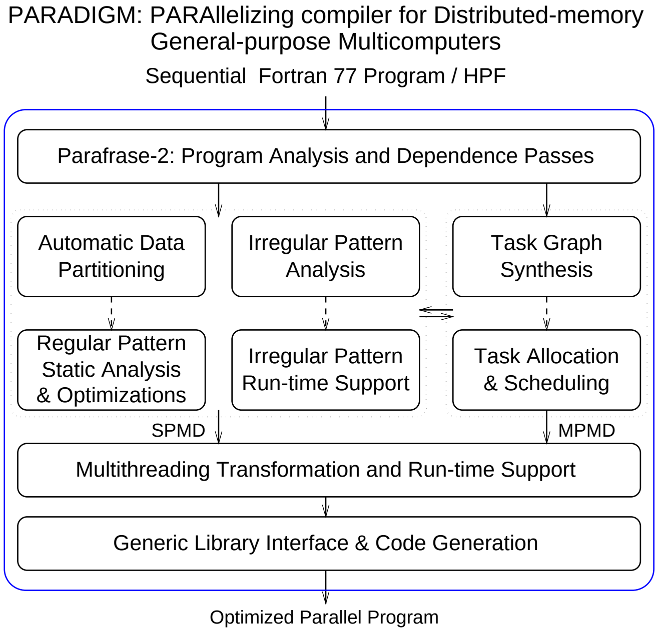


Figure 1.1: PARADIGM compiler overview

Fortran 90, supplying directives for specifying data partitioning and indicating parallelism. Also specified are a new language construct and library routines.

The information expressed in an HPF program is complementary to the PARADIGM compiler project. HPF allows the programmer to assist the compiler with the process of efficiently compiling code that may eventually be automated, as well as providing information that is impossible to determine at compile time or efficiently at run-time. Furthermore, HPF provides a language for the compiler to communicate with the programmer in expressing automatically chosen data partitioning.

1.3 Thesis Outline

Chapter 2 presents a brief summary of the High Performance Fortran language. A discussion of related work appears in Chapter 3. The implementation of HPF in

PARADIGM is detailed in Chapter 4. Chapter 5 presents an evaluation of the compiler using several benchmarks. Finally, conclusions and future work are discussed in Chapter 6. The Appendix contains the code generated from the benchmarks by PARADIGM.

2. HIGH PERFORMANCE FORTRAN

High Performance Fortran (HPF) is designed to be an extension to Fortran 90 that allows efficient and portable parallel code to be written for a large class of applications known as *data parallel*. The High Performance Fortran Forum (HPFF), consisting of industrial and academic groups, produced the first definition of HPF in 1993 and revised it in late 1994 [12]. Since then, several vendors have produced HPF compilers for a variety of parallel machines.

One of the most important features of HPF is the standardization of directives for data-parallel programming. The programmer is able to specify the layout of data at an abstract level and have the compiler generate the proper communication and local computation code. HPF also contains additional facilities for expressing parallelism and library routines often used in parallel programming.

As mentioned in [11], HPF is a rather complex language to fully implement. The following sections very briefly describe some of HPF, focusing on what is supported in the PARADIGM compiler project.

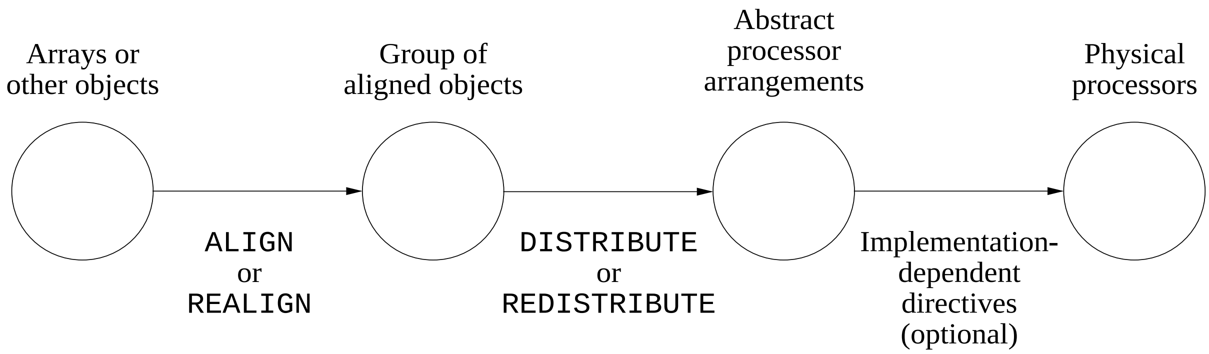


Figure 2.1: The HPF data mapping model

2.1 Data Mapping

The data alignment and distribution directives provided by HPF allow the programmer to inform the compiler how data arrays are to be partitioned. Thus, the programmer is still responsible for choosing partitionings that perform well but is freed from having to write code for the communication of data in the program.

2.1.1 Data-mapping model

Figure 2.1 shows a diagram of the HPF data-mapping model. Objects are first aligned with one another or themselves, if they are not aligned to something else. They are then distributed onto a rectilinear arrangement of abstract processors. The abstract processor arrangement is then mapped to physical processors. Each object is either aligned to another, explicitly distributed, or distributed according to the compiler's choice.

Objects are aligned and distributed according to specification directives immediately upon entry to a scoping unit. The executable directives affect the alignments and distribution as they are encountered in executing the body of code. The mapping of an object may change during execution by either redistributing the object to which it is aligned (which implicitly changes the object's mapping) or redistributing or realigning the object itself.

2.2 Directive Syntax

The syntax of HPF directives is consistent with that of Fortran 90. The directives begin with `!HPF$`, `CHPF$`, or `*HPF$`. However, `!HPF$` is preferred because it is compatible with both fixed and free source forms of Fortran. Furthermore, the directives themselves are defined only in free source form, where spaces are significant (unlike fixed source form).

There are two types of directives: *specification* and *executable*. Specification directives must appear in the declaration part of its containing program unit, whereas executable directives may appear where legal executable code may appear.

Finally, most directives have two forms: *statement* and *attribute*. The attribute form uses a "`::`" punctuation mark to allow multiple directives, according to certain rules, to be combined into one, such as to specify a template and its distribution using only one directive.

In the sections that follow, square brackets (`[ ]`) in the formal definitions of directives are used to indicate optional parts.

2.2.1 PROCESSORS directive

The `PROCESSORS` directive is a specification directive that is used to specify a rectilinear arrangement of processors describing an abstract machine. The processor arrangements defined can then be used in `DISTRIBUTE` directives to specify data distributions across processor configurations.

A processor arrangement name is a first-class, local entity and thus cannot have the same name as a variable, named constant, etc., defined in the same scoping unit. If multiple objects are mapped to the same abstract processor, they are mapped to the same physical processor at run-time. Processor arrangements cannot have extents of zero in any dimension. Finally, if two processor arrangements have the same shape, corresponding elements of the arrangements refer to the same abstract processor.

The intrinsics `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` may be used to obtain information on the number of physical processors, the rank, and extents at run-time.

The syntax of the `PROCESSORS` directive is

`!HPF$ PROCESSORS` *processors-decl-list*

or

`!HPF$ PROCESSORS ::` *processors-decl-list*

A *processors-decl-list* is

processors-name `[` *explicit-shape-list* `]`

An *explicit-shape-list* gives the rank of the abstract processor arrangement and its extent in each dimension.

The following are examples of using the `PROCESSORS` directive.

```
!HPF$ PROCESSORS ONE
!HPF$ PROCESSORS LINEAR(32)
!HPF$ PROCESSORS HYPER(2,2,2,2)
```

or equivalently

```
!HPF$ PROCESSORS :: ONE, LINEAR(32), HYPER(2,2,2,2)
```

The processor arrangement `ONE` describes a configuration of one-processor; `LINEAR` describes a one dimensional arrangement of 32 processors; and `HYPER` describes a four-dimensional ($2 \times 2 \times 2 \times 2$) arrangement of 16 processors. HPF requires that compilers support processor arrangements declared to have only a single processor and declared to contain a number of processors equal to the number of physical processors on which the program will execute (as returned by `NUMBER_OF_PROCESSORS`). Thus, on a machine with 12 processors, the compiler must support the first declaration, but support for the last two is implementation dependent.

2.2.2 DISTRIBUTE directive

The `DISTRIBUTE` directive is a specification directive that allows the programmer to specify directly a distribution for an object. Other objects may be aligned to this object. Two general kinds of distributions are allowed: *block* and *cyclic*. Block distributions specify that each processor gets a contiguous piece of the object. Cyclic distributions specify that each processor gets a piece of an object at regular intervals throughout the object. If the block size for a block distribution of a dimension is not specified by the programmer, the compiler uses a block size of $\lceil \frac{d}{p} \rceil$ for each dimension of the object, where d is the extent of the object in the dimension being considered, and p is the number of processors across which the object dimension is distributed. If a block size is not given with a cyclic distribution, it is one.

The syntax of the `DISTRIBUTE` directive is

`!HPF$ DISTRIBUTE` *distributee* `(`*dist-fmt-list*`)` `[ONTO` *processors-name*`]`

or

`!HPF$ DISTRIBUTE [(`*dist-fmt-list*`)]` `[ONTO` *processors-name*`]::`*distributee-list*

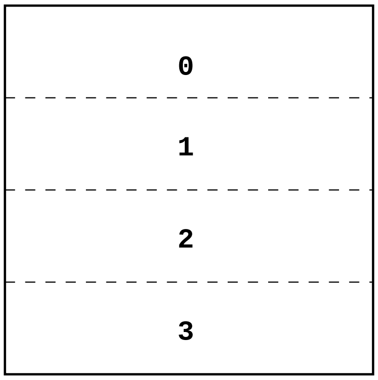
A *distributee* is an array or template. A *distributee-list* is a comma-separated list of *distributees.* A *processors-name* is an abstract processor arrangement declared in a `PROCESSORS` directive. The *dist-fmt-list* consists of a comma-separated list of the following:

- `BLOCK [`(int-expr)`]`, which indicates a block distribution of that dimension. The optional integer expression supplies the block size; otherwise, the block size is determined as mentioned previously.

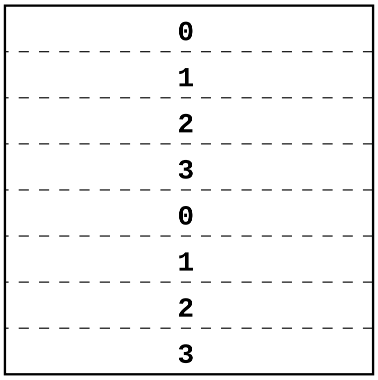
- `CYCLIC [`(int-expr)`]`, which indicates a cyclic distribution of that dimension. The optional integer expression supplies the block size; otherwise, the block size is one.

- "`*`", which indicates that the dimension is distributed "on-processor." Its use and an example are explained below.

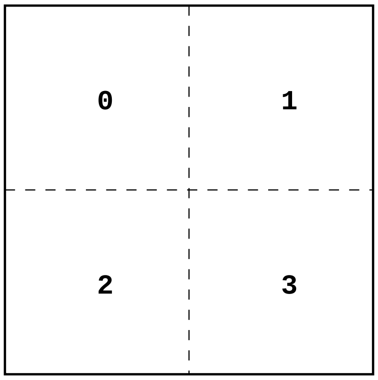
The `REDISTRIBUTE` directive syntax is similar. Figure 2.2 shows diagrams of several data distributions.



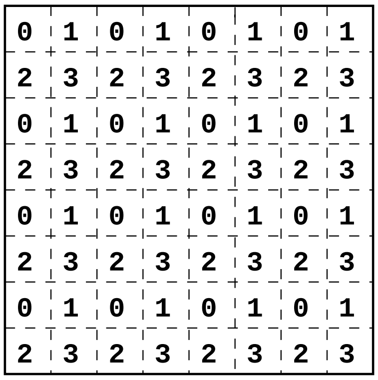
(a) (BLOCK, *)



(b) (CYCLIC(k), *)



(c) (BLOCK, BLOCK)



(d) (CYCLIC(k_1), CYCLIC(k_2))

Figure 2.2: Examples of data distributions for a two-dimensional array onto a four-processor arrangement

Non-unit block sizes in cyclic distributions result in *block-cyclic* distributions. Block-cyclic distributions give to each processor, at regular intervals, a contiguous piece of the object being distributed. In the code

```
      REAL A(64)
!HPF$ DISTRIBUTE A(CYCLIC(4))
```

the array A is distributed by giving each processor a four element block in round robin fashion.

The following directive specifies a block distribution for both dimensions of A:

```
      REAL A(64, 32)
!HPF$ DISTRIBUTE A(BLOCK, BLOCK)
```

If A is distributed onto a processor mesh of 4×4, each processor will receive a two-dimensional block of 16×8 elements of array A. If it is only one processor, the entire array is mapped to that processor.

To distribute objects onto lower dimensional processor arrangements, the "*" is used to specify that the corresponding dimension is distributed on-processor. Thus,

```
      REAL B(64, 32)
!HPF$ PROCESSORS LINE(4)
!HPF$ DISTRIBUTE B(BLOCK, *) ONTO LINE
```

distributes B by giving each processor 16 contiguous rows. The columns are distributed on-processor by the "*."

The distribution format may be omitted to specify that certain arrays be distributed on the same processor arrangement with the compiler choosing how to distribute them. In the code

```
!HPF$ DISTRIBUTE ONTO PROCS :: A, B, C
```

the arrays A, B, and C will have a distribution onto PROCS chosen by the compiler.

The REDISTRIBUTE directive is similar to the DISTRIBUTE directive except that the former is executable. Any object (array or template) that has been declared DYNAMIC may be redistributed (as long as it is not an *alignee*). The following is an example:

```
!HPF$ PROCESSORS LINE(4)
!HPF$ PROCESSORS SQUARE(2,2)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, *) ONTO LINE :: A
⋮
!HPF$ REDISTRIBUTE A(BLOCK, CYCLIC(4)) ONTO SQUARE
```

Note that the code above will cause communication to be generated by the compiler to redistribute the array A.

For dummy arguments to a subroutine, the DISTRIBUTE directive has three additional forms:

- The *prescriptive* form remaps an argument, if necessary, to the distribution described in the directive.
- The *transcriptive* form states that the subroutine will accept any mapping of the argument without remapping it.
- The *descriptive* form asserts that the argument already has the specified distribution and that no remapping is necessary.

For the details of dealing with dummy arguments, see [11].

2.2.3 ALIGN directive

The `ALIGN` directive is used to specify distributions of arrays relative to one another. It is used to efficiently group elements of arrays together on the same abstract processors. The `ALIGN` directive is a specification directive.

The syntax of the `ALIGN` directive follows. The `REALIGN` directive syntax is similar.

`!HPF$ ALIGN` *alignee* `(`*alignee-src-list*`) WITH` *align-target* `[(`*align-subs-list*`)]`

or

`!HPF$ ALIGN [(`*alignee-src-list*`)] WITH` *align-target* `[(`*align-subs-list*`)]::`*alignee-list*

An *alignee* is an array. An *align-target* is either an array or a template that is also distributed. An *alignee-list* is a comma-separated list of *alignees*. The *alignee-src-list* consists of a comma-separated list of the following:

- A ":" means that the elements of this dimension will be aligned with the corresponding dimension in the *align-subs-list*. The corresponding dimension is determined by matching colon-containing expressions in left-to-right order in the *alignee-src-list* and *align-subs-list*.
- A "*" indicates that the dimension is *collapsed*. A collapsed dimension means that the positions in that dimension are ignored in determining the corresponding alignment.
- An *align-dummy* is a dummy variable that has as its range all of the positions in that dimension.

An *align-subs-list* is a comma-separated list of:

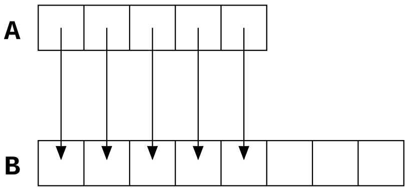
- *int-expr*, which is an integer expression not involving a dummy variable
- *linear-expr*, which is a linear function of a dummy variable
- *subscript-triplet*, which is the Fortran 90 construct
- "*", which indicates that the dimension is replicated

The complete details are beyond the scope of this discussion and can be found in [11].

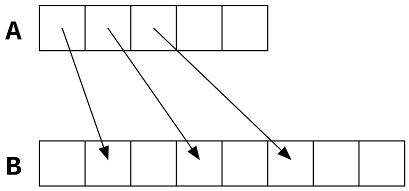
The following code shows examples of how the `ALIGN` directive is used. Figure 2.3 illustrates the alignments graphically.

```
!HPF$ ALIGN A1(:) WITH B1(:)
!HPF$ ALIGN A2(I) WITH B2(2*I)
!HPF$ ALIGN A3(I, J) WITH B3(J, I)
!HPF$ ALIGN A4(I, *) WITH B4(I)
!HPF$ ALIGN A5(*) WITH B5(*, *)
```

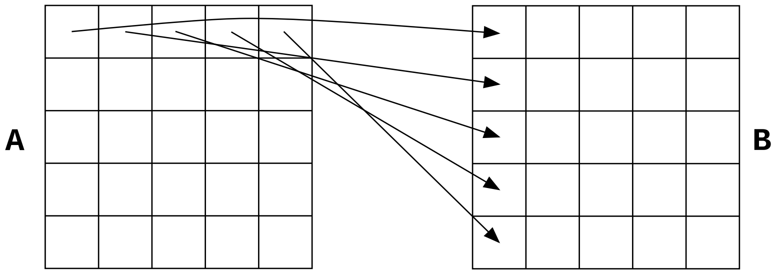
The first statement results in each element of `A1` being aligned with the corresponding element of `B1`. In the second statement, each element at position i in `A2` is aligned to the corresponding element at position $2i$ in `B2`. The third aligns the corresponding elements of the first dimension of `A3` with the elements of the second dimension of `B3`. Similarly, the elements of `A3`'s second dimension and `B3`'s first dimension are aligned. The fourth statement shows how to align a higher dimensional array with a lower dimensional object. The second dimension of `A4` is collapsed. The final statement replicates `A5` to be aligned with every element of `B5`. In practice, this means that `A5` is replicated on every abstract processor on which `B5` is distributed.



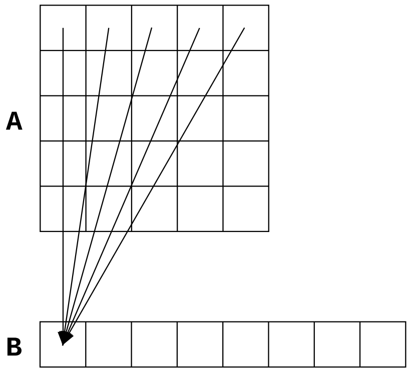
(a) `ALIGN A1(:)  WITH B1(:)`



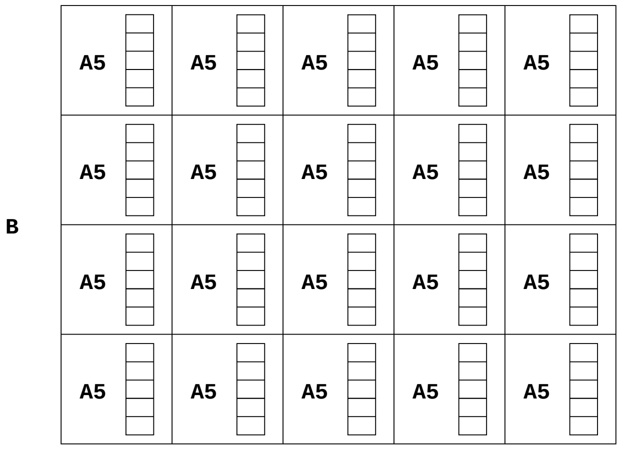
(b) `ALIGN A2(I) WITH B2(2*I)`



(c) `ALIGN A3(I, J) WITH B3(J, I)`



(d) `ALIGN A4(I, *) WITH B4(I)`



(e) `ALIGN A5(*) WITH B5(*, *)`

Figure 2.3: Examples of alignments

The `REALIGN` directive is similar to the `ALIGN` directive but is executable. It is used to change the relative mapping of one array to another. Realigning one array to another may cause elements of that array to be communicated and remapped, but it does not cause any other object to be remapped (unlike the `REDISTRIBUTE` directive).

The `ALIGN` directive has both *prescriptive* and *descriptive* forms whose definitions are analogous to those for the `DISTRIBUTE` directive. See [11] for more details.

2.2.4 TEMPLATE directive

The `TEMPLATE` directive is a specification directive used to declare templates that can then be used in `ALIGN` and `DISTRIBUTE` directives. A template name declaration in a directive must specify the rank and extent in each dimension.

A template is an abstract array used to aid in specifying data layouts. It is not allocated memory. A template name is a first-class, local entity and thus cannot have the same name as a variable, named constant, etc., defined in the same scoping unit. A template may be used as an *align-target* and be distributed.

The syntax of the `TEMPLATE` directive is

`!HPF$ TEMPLATE` *template-decl* `[,` ... `,` *template-decl*`]`

or

`!HPF$ TEMPLATE ::` *template-decl* `[,` ... `,` *template-decl*`]`

A *template-decl* is given by

template-name `[ (`*explicit-shape-list*`) ]`

An *explicit-shape-list* gives the rank of the template and its extent in each dimension. The `TEMPLATE` directive may also be, within certain constraints, combined with other directives using the "`::`" punctuation as shown in the examples below.

The following are examples of using `TEMPLATE` directives:

```
!HPF$ TEMPLATE LATE(25, 25)
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, CYCLIC), DIMENSION(64,32) :: A, B
```

The second directive shows how the "`::`" punctuation can be used to combine multiple directives.

Templates are especially useful in situations in which a programmer needs to align several arrays relative to one another, but declaring an array large enough to use as an *align-target* would be wasteful. The following example, from [11], shows how to align four $N \times N$ arrays to the four corners of a template of size $(N + 1) \times (N + 1)$ using the code

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1, N+1)
      REAL, DIMENSION(N, N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I, J) WITH EARTH( I ,  J )
!HPF$ ALIGN NE(I, J) WITH EARTH( I , J+1)
!HPF$ ALIGN SW(I, J) WITH EARTH(I+1,  J )
!HPF$ ALIGN SE(I, J) WITH EARTH(I+1, J+1)
```

Note that by using a `TEMPLATE` directive, there is no need to create an array just for this purpose.

2.2.5 DYNAMIC directive

The `DYNAMIC` directive is a specification directive used to specify that an object may be redistributed or realigned. It does not declare anything but adds the dynamic attribute to the object.

The syntax of the `DYNAMIC` directive is

!HPF$ DYNAMIC *object* [, ... , *object*]

or

!HPF$ DYNAMIC :: *object* [, ... , *object*]

The *objects* in a `DYNAMIC` directive must either be *alignees* or *distributees*. The `DYNAMIC` directive may also be, within certain constraints, combined with other directives.

The following code shows examples of using the `DYNAMIC` directive:

```
!HPF$ DYNAMIC A, B
!HPF$ DYNAMIC :: A, B
!HPF$ TEMPLATE, DYNAMIC, DISTRIBUTE(BLOCK), DIMENSION(10) :: A, B
```

The first two statements are equivalent, and the third shows how it may be combined with others.

2.2.6 INHERIT directive

The `INHERIT` directive is a specification directive which specifies that the template for a dummy argument is the same as that of the actual argument. The inherited template is not required to have the same size and shape as the dummy argument; the dummy argument will be aligned to the template in the same fashion as the actual argument is aligned to the template.

The syntax of the INHERIT directive is shown below.

!HPF$ INHERIT *dummy-argument-name*

Only dummy arguments can appear in an INHERIT directive, and dummy arguments in an INHERIT directive cannot also appear in an ALIGN directive.

An example of the INHERIT directive is as follows:

```
      SUBROUTINE SUBR(A)
      INTEGER A(16, 16)
!HPF$ INHERIT A
```

2.3 Data Parallelism

Data parallelism in HPF is expressed using the FORALL statement, the INDEPENDENT directive, and the PURE attribute. The FORALL statement is a language extension to Fortran 90 and does not appear as a comment. The INDEPENDENT directive is used to indicate that a loop is fully parallel. Both features will be discussed in the following sections.

The PURE attribute is also a language extension of Fortran 90. It is used in a function declaration to indicate that the function has no side effects on global data or on its arguments. The main use of pure functions is in FORALL statements and constructs.

2.3.1 FORALL statement and construct

The FORALL statement in HPF is provided to express data parallelism. It is an extension to the Fortran 90 language and is a generalization of the Fortran 90 array operations. Just like array operations, the FORALL statement conceptually computes all of the right-hand side expressions before performing any assignments. The `FORALL` statement, although it looks like a loop construct, is not a loop and does not define any ordered iterations. Rather, it defines an iteration space whose iterations can be evaluated in any order. Also, there is no non-determinancy in a `FORALL` statement. Thus, no two assignments to the same element are allowed.

The first form of the `FORALL` statement is

`FORALL (`*triplet*`, ... ,` *triplet* `[,` *mask*`])` *assignment*

This form is the *single-statement* `FORALL`. The *triplet* is an expression of the form

index-name `=` *lower bound* `:` *upper bound* `:` *stride*

The *mask* is a logical expression like that used in an `IF` statement; the *mask* is optional and is assumed true for all values if omitted.

The second and more general form of the `FORALL` statement is

```
FORALL (triplet, ... , triplet [, mask])
  body-stmt
    ⋮
  body-stmt
END FORALL
```

This form is the `FORALL` *construct.* The *body-stmt* is either an *assignment* statement, a `WHERE` statement, or another `FORALL` statement.

Because the PARADIGM compiler project accepts Fortran 77 plus HPF and some Fortran 90 array operations, the `WHERE` statement, defined in Fortran 90, will not be discussed further.

A simple example of a `FORALL` statement that cannot be expressed using only Fortran 90 array operations is

```
FORALL (I = 2:5) A(I, I) = A(I-1, I-1)
```

which results in the elements along the diagonal being shifted higher. Thus, if the diagonal elements were (11, 22, 33, 44, 55), they will become (11, 11, 22, 33, 44). Because all of the right-hand sides are evaluated before making any assignments, this code is *not* the same as the loop

```
DO I = 2, 5
  A(I, I) = A(I-1, I-1)
END DO
```

after which the diagonal would become (11, 11, 11, 11, 11).

In the following example, the mask is used to prevent division by zero.

```
FORALL (I = 1:10, J = 1:10, A(I, J) .NE. 0) A(I, J) = 1.0 / A(I, J)
```

The lower triangle of a matrix can be filled with zeros using the following code:

```
FORALL (I = 1:10, J = 1:10, I .LT. J) A(I, J) = 0.0
```

The single-statement `FORALL` is conceptually evaluated in four stages [11].

1. The *valid set* of index values is computed according to each *triplet*.

2. The *active set* of index values is determined by the *mask*. For each of the index values in the valid set, the *mask* is evaluated, and those not rejected by the *mask* become part of the active set.

3. For each tuple in the active set, the right-hand side of the statement in the `FORALL` is evaluated and saved, and any subexpressions in the left-hand side (such as array subscripts) are evaluated and saved.

4. Using the values for the right-hand side of the statement and left-hand side subexpressions, the assignment is carried out for each tuple in the active set.

The `FORALL` construct evaluation is similar. It is conceptually a sequence of single-statement `FORALL`s. For each assignment statement, it is executed using the process given previously. Each assignment statement is completed (including the actual assigning of values) before evaluating subsequent assignments. It is important to note that, just as before, the mask is evaluated before executing any of the body of the `FORALL`. If the `FORALL` is nested, the inner `FORALL` modifies the active set according to the rules previously discussed. The statements in its body are then evaluated with that active set. Upon exiting the inner `FORALL`, the active set reverts to what it was prior to entering the inner `FORALL`. Detailed examples of the `FORALL` construct can be found in [11].

2.3.2 INDEPENDENT directive

The `INDEPENDENT` directive is used by the programmer to indicate that the iterations of a `DO` loop or the active set tuples of a `FORALL` statement do not conflict with each other and may be executed in parallel. Thus, the iterations can be executed in any arbitrary order without changing the result. If the result is different with and without the `INDEPENDENT` directive, the code is not HPF-conforming, and the result is undefined.

The `INDEPENDENT` directive must immediately precede the `DO` or `FORALL` to which it applies. It is an assertion by the programmer that the result of the execution is the same regardless of the order and that the execution is not terminated by any statement within the DO or FORALL.

The INDEPENDENT directive also contains a NEW clause. NEW is used to declare variables as being private to the DO loop. In other words, each processor will have its own copy of the specified variables. It is necessary to include the index variables of inner DO loops in an enclosing NEW. Because the index names in FORALLs are already restricted by the FORALL semantics, they have no need of NEW.

The syntax of an INDEPENDENT directive is as follows:

```
!HPF$ INDEPENDENT [, NEW (variable,  ... , variable)]
```

Each *variable* declared in the NEW clause becomes private to the loop. The NEW clause is optional.

The following code shows a very simple use of the INDEPENDENT directive.

```
!HPF$ INDEPENDENT
      DO I = 1, 10
        A(I) = B(I-1) + B(I) + B(I+1)
      END DO
```

Although many compilers can detect that this loop is parallel, adding the directive ensures that it is always known to the compiler.

The next example shows how the INDEPENDENT directive is effective in assisting the compiler.

```
!HPF$ INDEPENDENT
      DO I = 1, 10
        A(INDX(I)) = B(I)
      END DO
```

By using the INDEPENDENT directive, the programmer asserts that the array INDX contains no repeated values. If it does, the code is non-conforming, and the result is undefined.

A final example illustrates a simple use of NEW.

```
!HPF$ INDEPENDENT, NEW(J)
      DO I = 2, 9
!HPF$ INDEPENDENT, NEW(KEQ)
        DO J = 2, 9
          KEQ = SQRT(B(I-1, J-1) + B(I, J) + B(I+1, J+1))
          A(I, J) = KEQ * B(I, J)
        END DO
      END DO
```

Here, the scalar KEQ can be used in the parallel loop since it is now private. Also, note that the index variable J must be private.

2.4 Intrinsic and Library Procedures

HPF defines a large number of new procedures to aid in writing data-parallel HPF code. Fortran 90 also has a large number of intrinsic functions and subroutines, all of which are included in HPF. The following sections describe only a few intrinsics. For more information on Fortran 90 intrinsics, see [13].

2.4.1 SUM and PRODUCT

The SUM intrinsic is defined by Fortran 90. Its usage is shown below.

```
SUM(ARRAY, DIM, MASK)
```

ARRAY is any array or section; DIM is the dimension to be summed and is optional; MASK is a Fortran 90 logical expression and is optional. If MASK is specified, only those element positions for which it is true are included in the summation. It returns a value which is the summation.

The `PRODUCT` intrinsic is defined by Fortran 90. Its usage is shown below.

```
PRODUCT(ARRAY, DIM, MASK)
```

The definition of `PRODUCT` is similar to that of `SUM` with the exception that `PRODUCT` multiplies the array elements.

2.4.2 MAXVAL and MINVAL

The `MAXVAL` intrinsic is defined by Fortran 90. Its usage is shown below.

```
MAXVAL(ARRAY, DIM, MASK)
```

`ARRAY` is any array or section; `DIM` is the dimension along which the maximum is to be determined and is optional; `MASK` is a Fortran 90 logical expression and is optional. If `MASK` is specified, only those element positions for which it is true are included in finding the maximum value. It returns a value which is the maximum value.

The `MINVAL` intrinsic is defined by Fortran 90. Its usage is shown below.

```
MINVAL(ARRAY, DIM, MASK)
```

The definition of `MINVAL` is similar to that of `MAXVAL` with the exception that `MINVAL` returns the minimum value of the array elements.

2.4.3 MAXLOC and MINLOC

The `MAXLOC` intrinsic is defined by Fortran 90. Its usage is shown below.

```
MAXLOC(ARRAY, DIM, MASK)
```

ARRAY is any array or section; DIM is the dimension along which the maximum is to be located and is optional; MASK is a Fortran 90 logical expression and is optional. If MASK is specified, only those element positions for which it is true are included in locating the maximum value. It returns an array whose contents specify the location of the maximum value.

The MINLOC intrinsic is defined by Fortran 90. Its usage is shown below.

```
MINLOC(ARRAY, DIM, MASK)
```

The definition of MINLOC is similar to that of MAXLOC with the exception that MINLOC locates the minimum value of the array elements.

3. RELATED WORK

High Performance Fortran [11] has been developed in a collaborative effort among researchers in industry and academia. Accordingly, HPF derives many of its features from academic and commercial projects. Also, a number of commercial HPF compilers are beginning to appear, including products from Applied Parallel Research, The Portland Group, Digital Equipment Corporation, and others. The following sections will briefly describe some of these research projects and commercial HPF compilers.

3.1 Other Fortran Language Extensions

Several extensions to Fortran for programming distributed-memory multicomputers have been designed. Research efforts include Fortran D, Fortran 90D, and Vienna Fortran. Among commercial extensions, CM Fortran also influenced HPF.

3.1.1 Fortran D

The Fortran D [14] language from Rice University is an extension to Fortran to allow compilers to compile Fortran efficiently for distributed-memory machines. It provides data-partitioning directives for the programmer to inform the compiler on how to partition the data. Through a combination of these directives, the programmer is able to specify how data are aligned with respect to each other and are distributed across processors, both statically and dynamically. As in HPF, the data may be distributed in a block, cyclic, or block-cyclic fashion in each dimension, and Fortran D provides irregular distributions by using mapping arrays. Fortran D also extends Fortran with a `FORALL` statement for expressing parallelism, although its definition is slightly different and also allows mapping iterations onto a particular set of processors.

3.1.2 Fortran 90D

The Fortran 90D [15] language from Syracuse University extends a subset of Fortran 90 with features from Fortran D. It includes the Fortran 90 array operations and a selected number of intrinsic functions.

3.1.3 Vienna Fortran

Vienna Fortran [16] is an extension of Fortran 77 designed to aid the programmer in writing efficient, portable code for a distributed-memory multicomputer. It is also much more complicated than the HPF language; for example, it allows alignments to form transitive relationships. As in HPF, the programmer writes code using global data references, and the compiler then makes the appropriate translation. Vienna Fortran provides facilities for explicitly distributing and aligning data, both statically and dynamically. Unlike the other languages mentioned here, all of the distribution information is represented by extending the Fortran language, rather than by using directives in comments. Other features of the language include the ability to specify very complex data mappings through the use of mapping arrays and the ability to explicitly map loops onto particular sets of processors.

3.1.4 CM Fortran

CM Fortran (CMF) [17] is a superset of Fortran 77 that includes the array operations and intrinsics from Fortran 90, alignment and data layout directives, and a single-statement `FORALL`. A set of simple alignment and layout directives allows the programmer to explicitly map data to processors. The single-statement `FORALL` is included as a generalization of Fortran 90 array operations and is defined like the single-statement `FORALL` in HPF. In CMF the parallelism is expressed explicitly through array operations and directives. The CMF compiler for the CM-5 compiles the language for an execution model in which a control processor executes all serial code and sends parallel code and data to a set of parallel processors.

3.2 Commercial HPF Compilers

Vendors are now beginning to supply HPF compilers. Currently, there are HPF compilers from Applied Parallel Research (APR), The Portland Group (PGI), and Digital Equipment Corporation (DEC), as well as others. The compilers from APR and PGI are targeted to multiple parallel machines from many vendors. Also, IBM is implementing HPF in their PTRAN compiler [18].

3.2.1 Applied Parallel Research

The APR product is called xHPF [19] and implements subset HPF plus a few additional HPF features; it also provides a set of its own custom directives. The xHPF compiler is actually a pre-compiler and is part of the FORGE environment. The compiler converts the HPF programs into a Fortran 77 program with message-passing calls. The code produced can then be compiled on a number of different machines with support from their run-time library.

3.2.2 The Portland Group

The PGHPF [20] compiler from PGI is a subset HPF compiler that also translates the source code to Fortran 77 (with a few extensions) plus message-passing calls. The output code is then compiled and linked with a run-time library. The PGHPF compiler is available for a very large number of parallel machines and also supports several standard message-passing libraries.

3.2.3 Digital Equipment Corporation

DEC, the company that organized the meeting at Supercomputing '91 that began defining HPF, claims to have the first full HPF implementation (with the exception of a very small number of unsupported features) with their DEC Fortran 90/HPF compiler [21]. The compiler attempts to generate code that is highly optimized after passing through several optimization phases that try to minimize communication and temporary arrays. The back end then performs many lower-level optimizations. It is targeted for high-speed networks of Alpha workstations as well as shared-memory, multiprocessor Alpha servers.

4. IMPLEMENTATION

In this chapter, the current implementation of HPF support for PARADIGM will be discussed.

4.1 Parafrase-2 Modifications

Because the PARADIGM compiler is built on top of Parafrase-2 (which accepts Fortran 77 and an extension called Cedar Fortran), it was necessary to modify Parafrase-2 to extend its capabilities. First, because HPF directives appear inside Fortran comments, the Parafrase-2 parser was modified to examine comments to determine if they are HPF directives. Because Fortran 77 continuations appear on subsequent lines, the HPF directive must be stored in a buffer as the parser continues to the next line to check for a continuation. When the parser parses the following line, the HPF directive(s) in the buffer are passed to the HPF directive parser.

The HPF directive parser was written using Flex [22] and Bison [23], because the directives themselves are defined in Fortran free source form. The parser accepts the entire set of HPF directives, although not all are currently supported in PARADIGM. The directives are parsed and stored in a private set of data structures. These data structures retain complete information about the directives parsed so that they may be generated in the output exactly as they were in the input.[1] The private data structures for the HPF parser use the Parafrase-2 symbol table; therefore, the objects declared in the directives are visible as normal.

The second set of modifications to Parafrase-2 allows it to accept the `FORALL` statement and construct. Since it already accepted a similar statement (from Cedar Fortran), the modifications required changes to the parser to accept a multi-statement construct and the addition of `END FORALL` to the keywords list, but no modifications to the Parafrase-2 data structures were required. With these modifications, Parafrase-2 is able to accept the HPF `FORALL` statement and construct and store it in its internal data structures. The conversion of each `FORALL` into Fortran 77 is performed in the HPF compiler pass.

The HPF directive parser performs a moderate amount of error checking, beyond syntax checking, to assist the programmer. It will issue errors for the following:

- using undeclared processor arrangements
- using an object other than a processor arrangement as a *processors-name*
- referencing undefined variables (that are not dummy variables)

[1]The formatting may change slightly because all directives are output in the attributed form.

- using a variable as an *alignee* and as an *align-target*
- having an incorrect number of dimensions in the *align-src-list* or *align-subs-list*
- using an incorrect number of dimensions in the *dist-fmt-list*
- having the number of dimensions not distributed “on-processor” not match the number of dimensions of the processor arrangement used in the directive

4.2 HPF Directive Conversion

As described, the HPF parser simply stores the information from the directives in private data structures. The information in the private data structures is then converted by the HPF compiler pass into PARADIGM's internal data structures after the entire program is parsed.

The HPF compiler pass converts the information contained in the `DISTRIBUTE` and `ALIGN` directives, in that order, into internal PARADIGM data structures. The `TEMPLATE` and `PROCESSORS` directives simply declare objects, and no conversion is necessary. The `INDEPENDENT` directive causes the source to be modified but otherwise does not require an explicit conversion. The remainder of HPF is not converted into PARADIGM's internal data structures because it is not yet supported.

The conversion of the `DISTRIBUTE` directive into PARADIGM's data structure is very simple. For each dimension, the type of distribution (block, cyclic, or “`*`”) and any expression for block size are stored. Also, if the distribution is onto a particular abstract processor arrangement, a pointer to the arrangement is stored.

To convert the `ALIGN` directive into PARADIGM's data structures, the rules given in the HPF language specification for matching dimensions must be followed. The rules are briefly as follows:

- Dimensions with colon-containing expressions are matched in a left-to-right order. Thus, a colon in an *alignee-src-list* can only match a colon or *subscript-triplet* in the *alignee-subs-list*. The dimensions that are matched are aligned as described by the expressions and must conform. There must be an equal number of colon-containing expressions in each list.

- A dimension with a dummy variable in the *alignee-src-list* matches the dimension in the *alignee-subs-list* with that dummy variable, possibly used in a linear expression. A dummy variable appearing in the *alignee-src-list*, but not in the other, is equivalent to a "*" (collapsed) for that dimension.

Note that neither collapsed nor replicated dimensions of the *alignee* are considered in determining which source and target dimensions are matched.

Following the rules stated above, the distribution information of the corresponding dimension of the *align-target* and the alignment expression, if any, are stored for each dimension of the *alignee*. If any dimension involves an expression and that expression is a linear expression containing a dummy variable, the stride and offset are extracted from the expression and stored in the data structure. Also, any explicit abstract processor arrangements from the distribution of the target are also stored.

The `INDEPENDENT` directive is not converted into any data structure, but it does cause source modifications. When applied to a `DO` loop, it becomes a `DOALL` loop and any variables in the `NEW` clause are declared private to that loop. In the case of a `FORALL` statement or construct, it prevents temporaries from being generated in the body when the compiler could not otherwise determine that they are not needed. Also, the directive asserts that the statements within an iteration must be executed in order, but no ordering among iterations is defined, further allowing optimizations.

In the course of performing conversions to PARADIGM's data structures, the HPF compiler pass detects and issues warnings or errors for the following:

- an attempt to distribute an *alignee* or having an object appear as an *alignee* more than once
- an *align-target* not appearing in a `DISTRIBUTE` directive
- an incorrect number of colon-containing expressions in an `ALIGN` directive

4.3 FORALL Conversion

The `FORALL` statement defined in HPF is a generalization of Fortran 90's array operations. Just as is the case for array operations, the right-hand side of a statement is completely evaluated for all index values before being assigned to the left-hand side. The efficient conversion of the FORALL statement into Fortran 77 DO loops in the PARADIGM compiler is discussed in this section.

In addition to handling the single-statement FORALL, the current implementation can handle the multi-statement FORALL construct if it is perfectly nested. If an imperfectly nested FORALL construct is present in the source code, it is passed through to the output unmodified and a warning is issued. The only other current limitation is that functions referenced in the index bounds with side effects on arrays used in the body are not handled correctly. These limitations compare very favorably with current commercial implementations.

When translating the FORALL into Fortran 77 DO loops, it is necessary to make sure that the semantics of the FORALL are preserved. Conceptually, the right-hand side of a statement in a FORALL is evaluated before the assignment to the left-hand side is made. Also, the bounds of the index names and the mask must be evaluated before the body. In practice, the goal is to optimize the translation to prevent, to the extent possible, temporary arrays and copying.

To preserve the FORALL semantics, the initial implementation will create a temporary array and copy the original array into it prior to the DO loop if it cannot prove that the semantics will not be violated. Because this implementation copies arrays, rather than attempting to evaluate the right-hand side into a temporary array, the temporary arrays are given the same distribution and alignment as the original array, and the resulting array is called a *clone*.

Using a clone, rather than evaluating the right-hand side of the statement into a temporary array and then performing the assignment, has the advantage of being able to use an efficient memory copy routine. For instance, the case of

```
FORALL (I = 1:100, X(I) .EQ. 0) X(I) = X(I-1) + X(I+1)
```

using the clone technique becomes

```
CALL MEM_COPY(X_CLONE, X, 100)
DO I = 1,100
  IF (X_CLONE(I) .EQ. 0)
    X(I) = X_CLONE(I-1) + X_CLONE(I+1)
  END IF
END DO
```

and evaluating into a temporary array becomes

```
DO I = 1,100
  IF (X(I) .EQ. 0)
    TMP(I) = X(I-1) + X(I+1)
  END IF
END DO
DO I = 1,100
  IF (X(I) .EQ. 0)
    X(I) = TMP(I)
  END IF
END DO
```

In the first version, the efficient memory copy is a system-supplied library routine (such as the C library function *memcpy*) that uses the best technique to copy the array. In the second case, each individual element of the array `X` is first examined, and then an element of the array `TMP` is copied into `X` if the condition is true. The second version is often slower because `X` is both read and written, and the array `TMP` is read. Even when the condition is false, every element of `X` must be read and tested individually.

There are cases in which the cloning technique is even more efficient relative to others, and there are cases in which it may actually be slower. Thus, the implementation could be optimized by providing some mechanism by which the better technique is chosen and applied.

Currently, PARADIGM receives as input and generates as output Fortran 77. In Fortran 77 there is no dynamic memory allocation; thus, reusing clones is very important. This implementation reuses clones as often as possible.

As clarified in version 1.1 of the HPF Language Specification, the `FORALL` index names have a scope local to the `FORALL` and do not conflict with any local or global entity of the same name. Thus, in converting a `FORALL`, the index names are replaced with unique names.

In converting the `FORALL` to a Fortran 77 `DO` loop (or nest of loops), the mask expression becomes the logical expression in an `IF` statement. So long as the semantics are preserved with the substitution of a clone when necessary, the transformation is valid.

Currently, when converting a `FORALL`, the compiler checks to see if the body is empty and issues a warning if the condition is true. Also, it performs a check to make sure that the body contains only valid statement types. Because only Fortran 77 is currently accepted, the only valid types are an assignment statement for a single-statement `FORALL` and assignment statements, other `FORALL`s, and `INDEPENDENT` directives for a multi-statement `FORALL` construct.

In translating multi-statement FORALL constructs to Fortran 77 DO loop nests, loop distribution may be used to preserve the semantics of the FORALL construct. There are other ways by which the semantics may be preserved, but distributing the FORALL construct has worked well in the initial implementation.

When distributing the FORALL into multiple DO loop nests and using the cloning technique described previously, one or more clones may be required to be used only in the bounds and mask expressions. In a FORALL construct, the bounds and mask must be evaluated and preserved in some manner before execution of the body. Because multiple DO loop nests may result in an array referenced in the bounds or a mask expression being modified before all DO loops are completed, one or more clones that persist for all the DO loop nests may be necessary.

To minimize the number of clones required in the multi-statement FORALL construct conversion, the compiler uses the dependence analysis from Parafrase-2 to determine if the FORALL will need to be split into multiple DO loop nests to preserve the FORALL semantics. In the case in which the dependence analysis can be used to prove that the semantics will not be violated, a single DO loop nest will be generated, and at most one clone for each array will be needed (which will be subsequently determined). If multiple DO loop nests will result, the bounds and mask expressions are checked to determine if there are dependence arcs to statements in the body which indicate that some array references in the expressions may be modified by the body. If such is the case, clones that persist for all DO loop nests will be created. Because the FORALL construct is distributed

```
CLONE(Loop)
CloneSet ← NIL
DepArcs ← DEPLIST(Loop)
while DepArcs ≠ φ
      do Arc ← NEXTARC(DepArcs)
         if DEPVEC(Arc) ≠ ⟨=, =, ..., =⟩
           then CloneSet ← CloneSet ∪ ARRAY(Arc)
         DepArcs ← DepArcs − Arc
INSERTCLONECODE(Loop, CloneSet)
```

Figure 4.1: Conservative clone minimization algorithm

into multiple `DO` loop nests when necessary, all copying for clones is performed outside of any loop nests.

As mentioned previously, Parafrase-2 dependence analysis is used to attempt to minimize the clones generated and the loops distributed. The algorithm to minimize the number of clones is shown in Figure 4.1. This algorithm is conservative but effective in this initial implementation. The source code

```
FORALL (I = 1:100, X(I) .NE. 0) X(I) = 1.0 / X(I)
```

is translated by this algorithm into

```
DO I = 1, 100
  IF (X(I) .NE. 0)
    X(I) = 1.0 / X(I)
  END IF
END DO
```

The algorithm correctly introduces no temporary arrays.

However, the source code

```
FORALL (I = 1:100) X(I) = X(I-1)
```

will be translated into

```
CALL MEM_COPY(X_CLONE, X, 100)
DO I = 1, 100
  X(I) = X_CLONE(I-1)
END DO
```

which does not require a clone. In this case, the clone may be removed if the order of the DO loop is reversed. However, the reversed DO loop really is implying a shift that must be handled efficiently by the compiler pass that generates SPMD code. These optimizations will be included in the compiler in the very near future.

The following example

```
FORALL (I = X(1):X(2), X(I-1) + X(I+1) .LE. 2) X(I) = X(I-1) + X(I+1)
```

is translated into

```
CALL MEM_COPY(X_CLONE, X, 100)
DO I = X_CLONE(1), X_CLONE(2)
  IF (X_CLONE(I-1) + X_CLONE(I+1) .LE. 2)
    X(I) = X_CLONE(I-1) + X_CLONE(I+1)
  END IF
END DO
```

by the current implementation, which executes efficiently. This example illustrates how the current implementation deals effectively with code requiring a temporary array.

The algorithm that attempts to minimize the number of DO loops generated from a multi-statement FORALL construct is shown in Figure 4.2. This algorithm is also conservative. As with the algorithm for inserting clones, this algorithm will soon be made more sophisticated.

The algorithm in Figure 4.2 results in the source code

```
FORALL (J = 1:100, I = 1:100, A(I, J) .NE. B(I, J))
  A(I, J) = A(I, J) / MAX
  B(I, J) = 1 / (B(I, J) - A(I, J))
END FORALL
```

```
Split(Loop, CurrStmt)
LastStmt ← next[CurrStmt]
while LastStmt ≠ NIL
        do DepArcs ← DepList(CurrStmt)
           while DepArcs ≠ ϕ
                 do Arc ← NextArc(DepArcs)
                    if (head[Arc] = LastStmt or tail[Arc] = LastStmt) and
                       DepVec(Arc) ≠ ⟨=, =, . . . , =⟩
                       then return LastStmt
                    DepArcs ← DepArcs − Arc
           LastStmt ← next[LastStmt]
return NIL
```

Figure 4.2: Conservative DO loop nest minimization algorithm

being translated into

```
DO J = 1, 100
  DO I = 1, 100
    IF (A(I, J) .NE. B(I, J))
      A(I, J) = A(I, J) / MAX
      B(I, J) = 1 / (B(I, J) - A(I, J))
    END IF
  END DO
END DO
```

The resulting source code correctly contains no temporary arrays.

Also, the algorithm translates the source code

```
FORALL (J = 2:99, I = 2:99, A(I, J) .NE. B(I, J))
  C(I, J) = SQRT(A(I, J) * A(J, I))
  B(I, J) = (C(I+1, J+1) + C(I-1, J-1)) / (B(I, J) - A(I, J))
END FORALL
```

into

```
DO J = 2, 99
  DO I = 2, 99
    IF (A(I, J) .NE. B(I, J))
      C(I, J) = SQRT(A(I, J) * A(J, I))
    END IF
```

```
    END DO
  END DO
  DO J = 2, 99
    DO I = 2, 99
      IF (A(I, J) .NE. B(I, J))
        B(I, J) = (C(I+1, J+1) + C(I-1, J-1)) / (B(I, J) - A(I, J))
      END IF
    END DO
  END DO
```

which results in two DO loop nests such that the FORALL semantics are preserved.

More optimization of the FORALL is necessary and will be implemented in the future by applying techniques used in vector compilation and extending them for distributed-memory multicomputers. Translation of the FORALL statement and construct into efficient code is important for HPF applications.

With the implementation of the FORALL statement in PARADIGM, Fortran 90 array operations may also be efficiently compiled using the same algorithms. The Fortran 90 array operations provide the programmer with a convenient and concise language for expressing the data-parallel array computations previously expressed with DO loops in Fortran 77.

Because the FORALL in HPF is a generalization of Fortran 90 array operations, translating a Fortran 90 array operation into a single-statement FORALL is straightforward. For example, the Fortran 90 finite difference code from [24]

```
    NEW(2:99, 2:99) = (X(1:98, 2:99) + X(3:100, 2:99) +
  &                    X(2:99, 1:98) + X(2:99, 3:100))/4
```

translates to

```
    FORALL (J = 2:99, I = 2:99)
  &   NEW(I, J) = (X(I-1, J) + X(I+1, J) + X(I, J-1) + X(I, J+1))/4
```

which can then be transformed into Fortran 77 by the techniques presented previously. In this case, the conversion to `DO` loops will not require any temporary arrays, and in fact none will be generated. If the left-hand side reference to `NEW` were replaced with an `X` reference, then one temporary array would be required.

Thus, a unified framework exists in PARADIGM for efficiently compiling both Fortran 90 array operations and `FORALL` statements.

4.4 Intrinsic and Library Procedures

As described in the previous chapter, HPF includes all of Fortran 90 and defines a large number of new intrinsic and library procedures and functions. In the initial implementation, only a few intrinsics are supported. More intrinsics will be implemented in the near future.

The `SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `MAXLOC`, and `MINLOC` intrinsic functions are defined in Fortran 90 and implemented here for parallel execution under HPF. To execute efficiently under HPF, the intrinsics must use the distribution information for their arguments. Because the PARADIGM compiler already efficiently compiles Fortran source code containing distributed data arrays [2]-[4], the intrinsics are implemented to take advantage of the compiler's ability to generate efficient code.

All of the intrinsics are implemented with inline source code created at compile time. Thus, there is no run-time library overhead. For each intrinsic, a nest of Fortran 77 `DO` loops is created that computes the appropriate value. Upon completion of the loops, the

```
      REAL A(64, 64), M
!HPF$ PROCESSORS SQUARE(4, 4)
!HPF$ DISTRIBUTE A(BLOCK, BLOCK) ONTO SQUARE

      M = SUM(A)
```

(a) Original code to obtain the sum of all elements of A

```
      REAL A(16, 16), M, LOCSUM, GLOBSUM

      LOCSUM = 0.0
      DO J = 1, 16
        DO I = 1, 16
          LOCSUM = LOCSUM + A(I, J)
        END DO
      END DO
      CALL MPI_ALLREDUCE(LOCSUM, GLOBSUM, 1, MPI_REAL, MPI_SUM,
     &                   MPI_COMM_WORLD, IERROR)
      M = GLOBSUM
```

(b) Generated SPMD code to compute local and then global sum

Figure 4.3: Example of translating a use of the SUM intrinsic to SPMD code

local value is efficiently combined with other local values to determine and subsequently broadcast the global value. Efficient methods for implementing the computation of a global value from a local value are often available in message-passing libraries. Shown in Figure 4.3 is an example of this translation using the MPI [25] message-passing library.

5. RESULTS

In this chapter, performance results for several benchmarks are reported. The benchmark codes are intended to show how the PARADIGM compiler is able to use the HPF directives to generate SPMD message-passing code. Each benchmark was run on an Intel Paragon and an Ethernet-based network of Sun SPARCstation 5 workstations running SunOS 4.1.3.

5.1 Jacobi's Iterative Method

Jacobi's iterative method, illustrated in Figure 5.1, computes the numerical solution of the elliptic partial differential equation known as Poisson's equation. The loop of the computation can be expressed succinctly as either an array operation or the single-statement `FORALL` used here. The distribution used for the data arrays in the source is shown in Figure 5.2. The original source code is shown in Figure 5.3, and the source code generated by PARADIGM is shown in Appendix A. Performance results are shown in Figures 5.4 and 5.5.

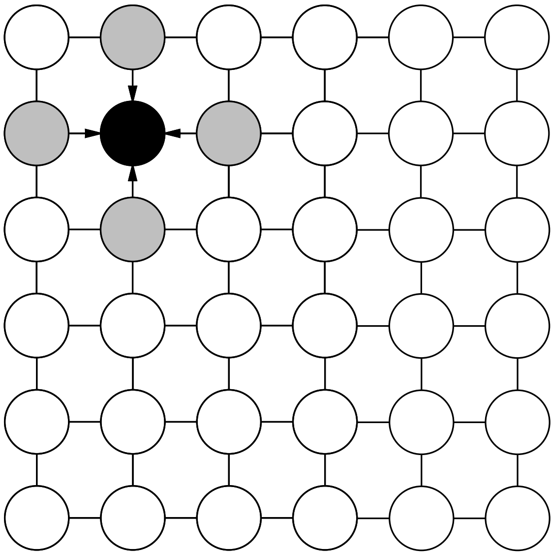


Figure 5.1: Illustration of Jacobi

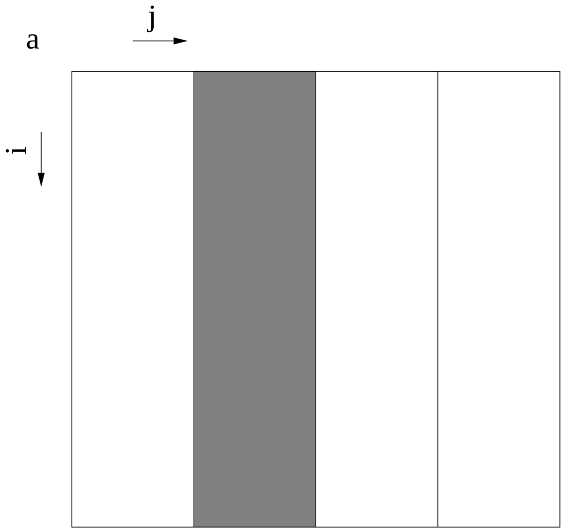


Figure 5.2: Distribution of Jacobi

```
        program jacobi

      integer i, j, k
      real a(1001, 1001)
!HPF$ processors line(4)
!HPF$ distribute a(*, block) onto line

      do k = 1, 100
        forall (j = 2:999, i = 2:999)
          a(i, j) = (a(i - 1, j) + a(i + 1, j) + a(i, j - 1) + a(i,
     *                j + 1)) / 4
        end forall
      end do

      end
```

Figure 5.3: Jacobi's iterative method

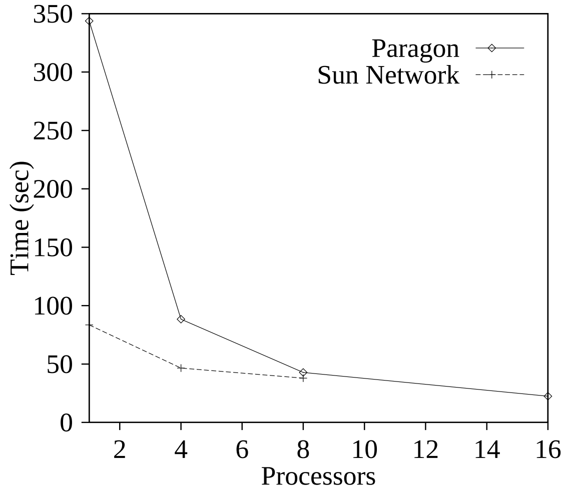


Figure 5.4: Time for Jacobi

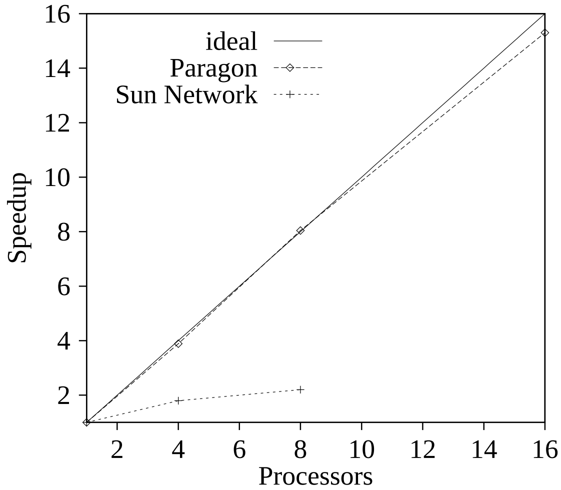


Figure 5.5: Speedup for Jacobi

5.2 Reduction

The reduction benchmark computes values for the array `q` and then uses the `SUM` intrinsic to compute the sum of `q`'s elements. Figure 5.6 shows the original source code, and the source code generated by PARADIGM is shown in Appendix A. Performance results are shown in Figures 5.7 and 5.8.

5.3 Simple Matrix Multiplication

Matrix multiplication is performed in this benchmark by computing the middle product, as illustrated in Figure 5.9. This simple form of matrix multiplication is easy to express with a `FORALL` statement nested in a `DO` loop. In the first version of the code, `a` and `b` are partially replicated so that a block of `c` is computed on each processor. The second version of the code gives `a`, `b`, and `c` a two-dimensional block distribution. For the code using replication, Figure 5.10 shows the original source code, and performance

```
      program reduction

      integer i
      real s, a(100000), b(100000), c(100000), q(100000)
!HPF$ processors line(4)
!HPF$ distribute q(block) onto line
!HPF$ align with q :: a, b, c

      forall (i = 1:100000) q(i) = a(i) * b(i) / c(i)
      s = sum(q)
      end
```

Figure 5.6: Array summation using SUM

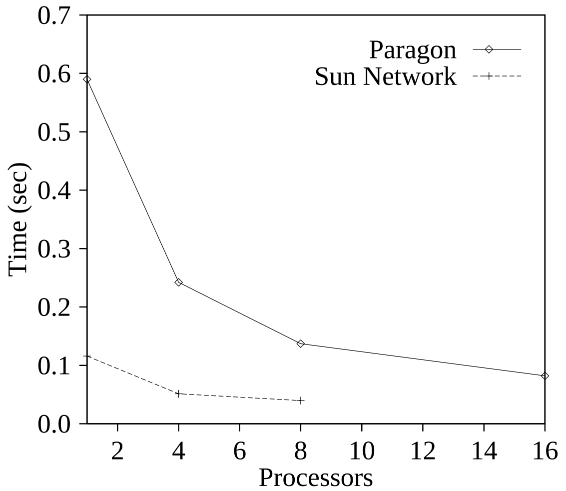


Figure 5.7: Time for reduction

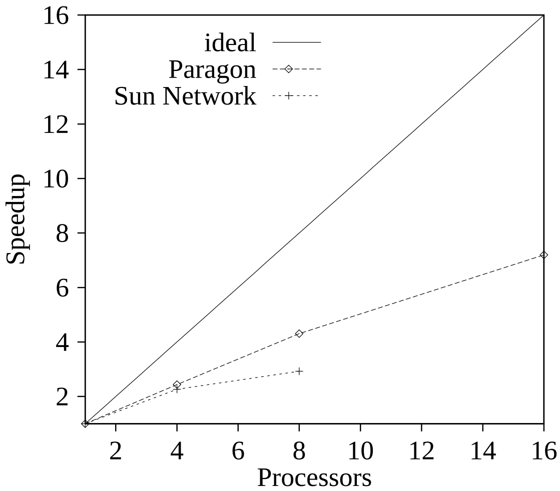


Figure 5.8: Speedup of reduction

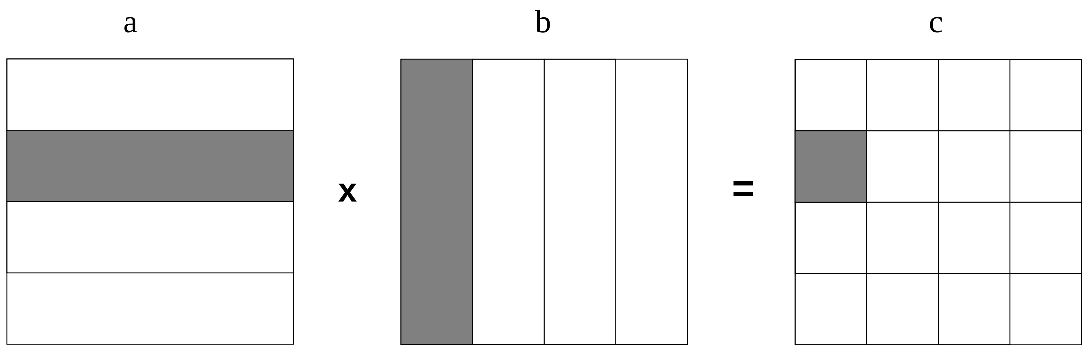


Figure 5.9: Illustration of matrix multiplication

results are shown in Figures 5.11 and 5.12. For the code using block distribution, the original source code is shown in Figure 5.13, and Figures 5.14 and 5.15 show the performance results. The source codes generated by PARADIGM for both versions are shown in Appendix A.

5.4 Two-Dimensional Explicit Hydrodynamics Kernel

The two-dimensional explicit hydrodynamics kernel from the set of Livermore kernels [26] is the main set of computation loops extracted from a hydrodynamics application. The computation is naturally expressed using the data-parallel `FORALL` construct. The original source code is shown in Figure 5.16, and the source code generated by PARADIGM is shown in Appendix A. Figures 5.17 and 5.18 show the performance results.

```
      program matmul

      real a(256, 256), b(256, 256), c(256, 256)
      integer i, j, k
!HPF$ processors p(2, 2)
!HPF$ distribute c(block, block) onto p
!HPF$ align a(i, *) with c(i, *)
!HPF$ align b(*, i) with c(*, i)

      forall (j = 1:256, i = 1:256) c(i,j) = 0
      do j = 1,256
        do k = 1,256
          forall(i = 1:256) c(i, j) = c(i, j) + a(i, k) * b(k, j)
        end do
      end do

      end
```

Figure 5.10: Simple matrix multiplication (replication)

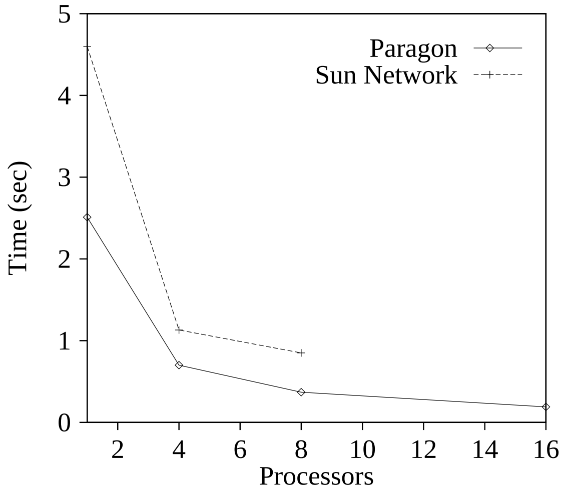


Figure 5.11: Time for matrix multiplication (replication)

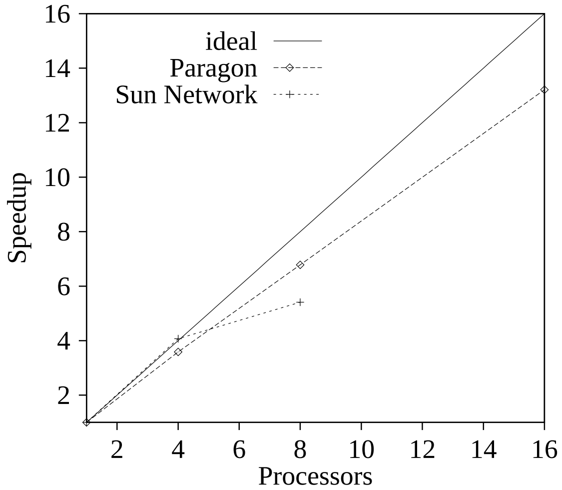


Figure 5.12: Speedup of matrix multiplication (replication)

```
      program matmul

      real a(256, 256), b(256, 256), c(256, 256)
      integer i, j, k
!HPF$ processors p(2, 2)
!HPF$ distribute c(block, block) onto p
!HPF$ align with c :: a, b

      forall (j = 1:256, i = 1:256) c(i,j) = 0
      do j = 1,256
        do k = 1,256
          forall(i = 1:256) c(i, j) = c(i, j) + a(i, k) * b(k, j)
        end do
      end do

      end
```

Figure 5.13: Simple matrix multiplication (block distribution)

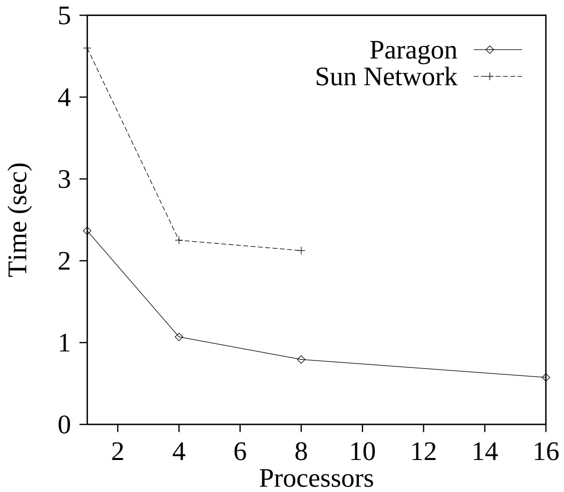


Figure 5.14: Time for matrix multiplication (block distribution)

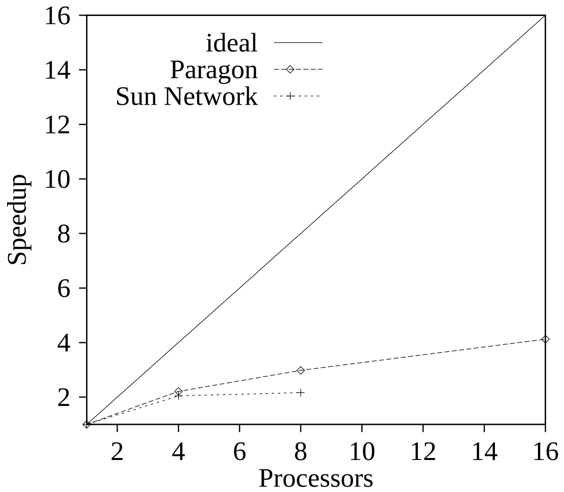


Figure 5.15: Speedup of matrix multiplication (block distribution)

```
      program explhydro

      real za(1000, 7), zr(1000, 7), zb(1000, 7), zu(1000, 7),
     *      zz(1000,7), zv(1000, 7), zq(1000, 7), zm(1000, 7),
     *      zp(1000, 7)
!HPF$ processors line(4)
!HPF$ distribute za(block, *) onto line
!HPF$ align with za :: zb, zm, zp, zq, zr, zu, zv, zz

      do l = 1, 100
        t = 0.0037
        s = 0.0041

        forall (k=2:6, j=2:999)
          za(j, k) = (zp(j-1, k+1) + zq(j-1, k+1) - zp(j-1, k) -
     * zq(j-1, k)) * (zr(j, k) + zr(j-1, k)) / (zm(j-1, k) +
     * zm(j-1, k+1))
          zb(j, k) = (zp(j-1, k) + zq(j-1, k) - zp(j, k) -
     * zq(j, k) * (zr(j, k) + zr(j, k-1)) / zm(j, k) + zm(j-1, k))
        end forall

        forall (k=2:6, j=2:999)
          zu(j, k) = zu(j, k) + s * (za(j, k) * (zz(j, k) - zz(j+1
     * , k)) - za(j-1, k) * (zz(j, k) - zz(j-1, k)) - zb(j, k) * (zz(
     * j, k) - zz(j, k-1)) + zb(j, k+1) * (zz(j, k) - zz(j, k+1)))
          zv(j, k) = zv(j, k) + s * (za(j, k) * (zr(j, k) - zr(j+1
     * , k)) - za(j-1, k) * (zr(j, k) - zr(j-1, k)) - zb(j, k+1) *
     * (zr(j, k) - zr(j, k-1)) + zb(j, k+1) * (zr(j, k) - zr(j, k+1)))
        end forall

        forall (k=2:6, j=2:999)
          zr(j, k) = zr(j, k) + t * zu(j, k)
          zz(j, k) = zz(j, k) + t * zv(j, k)
        end forall
      enddo

      end
```

Figure 5.16: Two-dimensional explicit hydrodynamics Livermore kernel

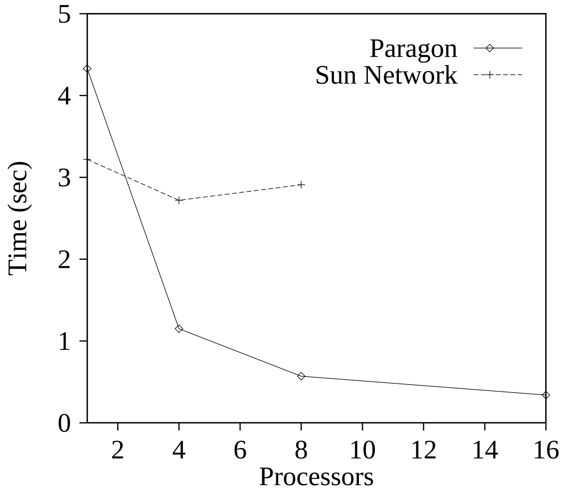


Figure 5.17: Time for two-dimensional explicit hydrodynamics kernel

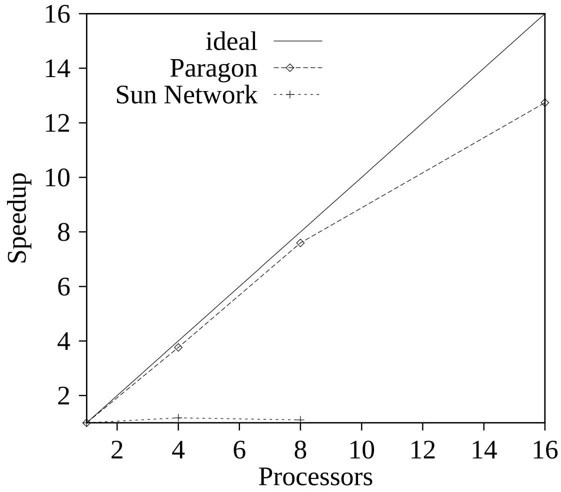


Figure 5.18: Speedup of two-dimensional explicit hydrodynamics kernel

6. CONCLUSIONS

Parallel computing has lacked a portable, efficient environment for writing parallel applications. High Performance Fortran provides such an environment for the programmer writing data-parallel applications. The work in this thesis implements HPF support in the PARADIGM compiler. Research in the PARADIGM compiler project seeks to alleviate the programmer's burden of writing explicitly parallel applications using non-portable libraries and language extensions. The addition of HPF will help further this research and aid the compiler in parallelizing data-parallel applications.

Imperfectly nested `FORALL`s will be supported in the future, and the translation of `FORALL`s to efficient Fortran 77 will include more aggressive optimizations. Also, as the compiler is better able to handle `FORALL`s, more efficient code will be generated from the Fortran 90 array operations.

There are a large number of intrinsic functions in Fortran 90 and library procedures in HPF. Additional work will include implementing selected routines for efficient execution on distributed-memory multicomputers. As with the current implementation, the intrinsics will be generated by the compiler, rather than being contained in a run-time library.

Finally, the generation of efficient SPMD message-passing Fortran source code continues to be an important research area. As the compiler is able to generate better code, the code generated by this work will benefit.

APPENDIX A. PARADIGM GENERATED CODE

Presented in this appendix are the message-passing codes generated by PARADIGM. Only the four-processor MPI version is shown for each benchmark.

Jacobi:

```
PROGRAM jacobi
include 'mpif.h'
CHARACTER m$buf(1)
INTEGER my$host, my$proc, m$numproc, m$status
DOUBLE PRECISION m$retval
DOUBLE PRECISION t$data
INTEGER my$p(2)
INTEGER m$num(2)
INTEGER m$numdim
INTEGER m$to(-1:1,-1:1)
INTEGER cln_0
INTEGER cln
REAL a_clone(1001,0:252)
INTEGER idx_2
INTEGER idx_1
INTEGER i, j, k
REAL a(1001,0:252)
DOUBLE PRECISION globsum,t

CALL MPI_INIT(m$status)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,my$proc,m$status)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,m$numproc,m$status)
CALL MPI_BARRIER(MPI_COMM_WORLD,m$status)
```

```
      t$data = MPI_WTIME()
      m$numdim = 2
      m$num(1) = 4
      m$num(2) = 1
      my$p(1) = mod(my$proc,m$num(1))
      my$p(2) = my$proc / m$num(1)
      IF ((my$p(2) .GE. m$num(2)) .OR. (my$p(1) .GE. m$num(1))) THEN
        t$data = MPI_WTIME() - t$data
        PRINT *, my$proc, t$data
        CALL MPI_FINALIZE(m$status)
        STOP
      END IF
      m$to( -1,0) = m$num(1) * my$p(2) + my$p(1) - 1
      m$to(1,0) = m$num(1) * my$p(2) + my$p(1) + 1
      DO k = 1,100
        CALL MEMCPY(a_clone,a,1013012)
        IF (my$p(1) .GE. 1) THEN
          CALL MPI_SEND(a_clone(1,1),4004,MPI_BYTE,m$to( -1,0),0,MPI_C
     1    OMM_WORLD,m$status)
        END IF
        IF (my$p(1) .LE. 2) THEN
          CALL MPI_RECV(a_clone(1,252),4004,MPI_BYTE,m$to(1,0),0,MPI_CO
     1    MM_WORLD,m$retval,m$status)
        END IF
        IF (my$p(1) .LE. 2) THEN
          CALL MPI_SEND(a_clone(1,251),4004,MPI_BYTE,m$to(1,0),1,MPI_C
     1    OMM_WORLD,m$status)
        END IF
        IF (my$p(1) .GE. 1) THEN
          CALL MPI_RECV(a_clone(1,0),4004,MPI_BYTE,m$to( -1,0),1,MPI_CO
     1    MM_WORLD,m$retval,m$status)
        END IF
        DO idx_1 = (MAX(( -251) * my$p(1) + 2,1)),(MIN(( -251) * my$p
     1    (1) + 999,251))
          DO idx_2 = 2,999
            a(idx_2,idx_1) = (a_clone(idx_2 - 1,idx_1) + a_clone(idx_2
     1      + 1,idx_1) + a_clone(idx_2,idx_1 - 1) + a_clone(idx_2,idx_1
     2      + 1)) / 4
          END DO
        END DO
      END DO
      t$data = MPI_WTIME() - t$data
      PRINT *, my$proc, t$data
      CALL MPI_FINALIZE(m$status)
      END
```

Reduction:

```
PROGRAM reduction
include 'mpif.h'
IMPLICIT NONE
CHARACTER m$buf(1)
INTEGER my$host, my$proc, m$numproc, m$status
DOUBLE PRECISION m$retval
DOUBLE PRECISION t$data
INTEGER my$p(2)
INTEGER m$num(2)
INTEGER m$numdim
INTEGER m$to(-1:1,-1:1)
REAL locsum
INTEGER scn_0
INTEGER scn
INTEGER idx_2
INTEGER idx_1
INTEGER idx_0
INTEGER idx
INTEGER i
REAL s
REAL a(25000,1)
REAL b(25000,1)
REAL c(25000,1)
REAL q(25000,1)
INTEGER j
REAL globsum

CALL MPI_INIT(m$status)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,my$proc,m$status)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,m$numproc,m$status)
CALL MPI_BARRIER(MPI_COMM_WORLD,m$status)
t$data = MPI_WTIME()
m$numdim = 2
m$num(1) = 4
m$num(2) = 1
my$p(1) = mod(my$proc,m$num(1))
my$p(2) = my$proc / m$num(1)
IF ((my$p(2) .GE. m$num(2)) .OR. (my$p(1) .GE. m$num(1))) THEN
  t$data = MPI_WTIME() - t$data
  PRINT *, my$proc, t$data
  CALL MPI_FINALIZE(m$status)
```

```
        STOP
      END IF
      DO idx_2 = 1,1
        DO idx_1 = (1),(25000)
          q(idx_1,idx_2) = a(idx_1,idx_2) * b(idx_1,idx_2) / c(idx_1,id
     1    x_2)
        END DO
      END DO
      locsum = 0.0
      DO scn = 1,1
        DO scn_0 = (1),(25000)
          locsum = locsum + q(scn_0,scn)
        END DO
      END DO
      CALL MPI_ALLREDUCE(locsum, globsum, 1, MPI_REAL, MPI_SUM,
     & MPI_COMM_WORLD, m$status)
      s = globsum
      t$data = MPI_WTIME() - t$data
      PRINT *, my$proc, t$data
      CALL MPI_FINALIZE(m$status)
      END
```

Matrix multiplication (replication):

```
      PROGRAM matmul
      include 'mpif.h'
      CHARACTER m$buf(1)
      INTEGER my$host, my$proc, m$numproc, m$status
      DOUBLE PRECISION m$retval
      DOUBLE PRECISION t$data
      INTEGER my$p(2)
      INTEGER m$num(2)
      INTEGER m$numdim
      INTEGER m$to(-1:1,-1:1)
      INTEGER procs(2,2)
      INTEGER idx_1, idx_0, idx, i, j, k
      REAL a(128,256)
      REAL b(256,128)
      REAL c(128,128)

      CALL MPI_INIT(m$status)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,my$proc,m$status)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,m$numproc,m$status)
      CALL MPI_BARRIER(MPI_COMM_WORLD,m$status)
```

```
t$data = MPI_WTIME()
m$numdim = 2
m$num(1) = 2
m$num(2) = 2
my$p(1) = mod(my$proc,m$num(1))
my$p(2) = my$proc / m$num(1)
IF ((my$p(2) .GE. m$num(2)) .OR. (my$p(1) .GE. m$num(1))) THEN
  t$data = MPI_WTIME() - t$data
  PRINT *, my$proc, t$data
  CALL MPI_FINALIZE(m$status)
  STOP
END IF
DO idx_0 = (1),(128)
  DO idx = (1),(128)
    c(idx,idx_0) = 0
  END DO
END DO
DO j = (1),(128)
  DO k = 1,256
    DO idx_1 = (1),(128)
      c(idx_1,j) = c(idx_1,j) + a(idx_1,k) * b(k,j)
    END DO
  END DO
END DO
t$data = MPI_WTIME() - t$data
PRINT *, my$proc, t$data
CALL MPI_FINALIZE(m$status)
END
```

Matrix multiplication (block distribution):

```
PROGRAM matmul
include 'mpif.h'
INTEGER m$ptr
REAL m$buf(65540)
INTEGER my$host, my$proc, m$numproc, m$status
DOUBLE PRECISION m$retval
DOUBLE PRECISION t$data
INTEGER my$p(2)
INTEGER m$num(2)
INTEGER m$numdim
INTEGER m$to(-1:1,-1:1)
INTEGER m$m2
INTEGER m$n2
```

```
      INTEGER m$q2
      INTEGER m$p2
      INTEGER m$m1
      INTEGER m$n1
      INTEGER m$q1
      INTEGER m$p1
      INTEGER idx_1, idx_0, idx, i, j, k
      REAL a(256,256)
      REAL b(256,256)
      REAL c(256,256)

      CALL MPI_INIT(m$status)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,my$proc,m$status)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,m$numproc,m$status)
      CALL MPI_BARRIER(MPI_COMM_WORLD,m$status)
      t$data = MPI_WTIME()
      m$numdim = 2
      m$num(1) = 2
      m$num(2) = 2
      my$p(1) = my$proc / m$num(1)
      my$p(2) = mod(my$proc,m$num(1))
      IF ((my$p(2) .GE. m$num(2)) .OR. (my$p(1) .GE. m$num(1))) THEN
        t$data = MPI_WTIME() - t$data
        PRINT *, my$proc, t$data
        CALL MPI_FINALIZE(m$status)
        STOP
      END IF
      m$p1 = my$p(1)
      m$p2 = my$p(2)
      DO idx_0 = (1 + (128 * m$p2)),(MIN(256,128 * (1 + m$p2)))
        DO idx = (1 + (128 * m$p1)),(MIN(256,128 * (1 + m$p1)))
          c(idx,idx_0) = 0
        END DO
      END DO
      m$q1 = my$p(1)
      m$q2 = my$p(2)
      DO m$p1 = 0,1
        DO m$p2 = 0,1
          IF ((m$p1 .NE. m$q1) .OR. (m$p2 .NE. m$q2)) THEN
            m$ptr = 1
            DO k = 1 + (128 * m$q2),MIN(256,128 * (1 + m$q2))
              DO idx_1 = MAX(1 + (128 * m$p1),1 + (128 * m$q1)),MIN(2
     1        56,128 * (1 + m$p1),128 * (1 + m$q1))
                m$buf(m$ptr) = a(idx_1,k)
                m$ptr = m$ptr + 1
```

```
          END DO
         END DO
         DO j = MAX(1 + (128 * m$p2),1 + (128 * m$q2)),MIN(256,128 *
     1    (1 + m$p2),128 * (1 + m$q2))
          DO k = 1 + (128 * m$q1),MIN(256,128 * (1 + m$q1))
           m$buf(m$ptr) = b(k,j)
           m$ptr = m$ptr + 1
          END DO
         END DO
         CALL MPI_SEND(m$buf,m$ptr*4,MPI_BYTE,m$p1*2 + m$p2,4096*
     1   m$q1 + 64 * m$q2,MPI_COMM_WORLD,m$status)
        END IF
       END DO
      END DO
      m$p1 = my$p(1)
      m$p2 = my$p(2)
      DO m$q1 = 0,1
       DO m$q2 = 0,1
        IF ((m$p1 .NE. m$q1) .OR. (m$p2 .NE. m$q2)) THEN
         m$ptr = 1
         CALL MPI_RECV(m$buf,65540,MPI_BYTE,m$q1 * 2 + m$q2,
     1   4096 * m$q1 + 64 * m$q2,MPI_COMM_WORLD,m$retval,m$status)
         DO k = 1 + (128 * m$q2),MIN(256,128 * (1 + m$q2))
          DO idx_1 = MAX(1 + (128 * m$p1),1 + (128 * m$q1)),MIN(2
     1    56,128 * (1 + m$p1),128 * (1 + m$q1))
           a(idx_1,k) = m$buf(m$ptr)
           m$ptr = m$ptr + 1
          END DO
         END DO
         DO j = MAX(1 + (128 * m$p2),1 + (128 * m$q2)),MIN(256,128 *
     1    (1 + m$p2),128 * (1 + m$q2))
          DO k = 1 + (128 * m$q1),MIN(256,128 * (1 + m$q1))
           b(k,j) = m$buf(m$ptr)
           m$ptr = m$ptr + 1
          END DO
         END DO
        END IF
       END DO
      END DO
      DO j = (1 + (128 * m$p2)),(MIN(256,128 * (1 + m$p2)))
       DO k = (1),(256)
        DO idx_1 = (1 + (128 * m$p1)),(MIN(256,128 * (1 + m$p1)))
         c(idx_1,j) = c(idx_1,j) + a(idx_1,k) * b(k,j)
        END DO
       END DO
```

```
END DO
t$data = MPI_WTIME() - t$data
PRINT *, my$proc, t$data
CALL MPI_FINALIZE(m$status)
END
```

Two-dimensional explicit hydrodynamics kernel (from Livermore Kernels):

```
PROGRAM explhydro
include 'mpif.h'
CHARACTER m$buf(84)
INTEGER my$host, my$proc, m$numproc, m$status
DOUBLE PRECISION m$retval
DOUBLE PRECISION t$data
INTEGER my$p(2)
INTEGER m$num(2)
INTEGER m$numdim
INTEGER m$to(-1:1,-1:1)
INTEGER idx_6
INTEGER idx_5
INTEGER idx_4
INTEGER idx_3
INTEGER idx_2
INTEGER idx_1
INTEGER idx_0
INTEGER idx
REAL za(0:250,7)
REAL zr(0:251,7)
REAL zb(250,7)
REAL zu(250,7)
REAL zz(0:251,7)
REAL zv(250,7)
REAL zq(0:250,7)
REAL zm(0:250,7)
REAL zp(0:250,7)
INTEGER i, j, l
REAL t, s
INTEGER k

CALL MPI_INIT(m$status)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,my$proc,m$status)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,m$numproc,m$status)
CALL MPI_BARRIER(MPI_COMM_WORLD,m$status)
t$data = MPI_WTIME()
```

```
      m$numdim = 2
      m$num(1) = 4
      m$num(2) = 1
      my$p(1) = mod(my$proc,m$num(1))
      my$p(2) = my$proc / m$num(1)
      IF ((my$p(2) .GE. m$num(2)) .OR. (my$p(1) .GE. m$num(1))) THEN
        t$data = MPI_WTIME() - t$data
        PRINT *, my$proc, t$data
        CALL MPI_FINALIZE(m$status)
        STOP
      END IF
      m$to( -1,0) = m$num(1) * my$p(2) + my$p(1) - 1
      m$to(1,0) = m$num(1) * my$p(2) + my$p(1) + 1
      IF (my$p(1) .LE. 2) THEN
        CALL f$pack2(zm,4,0,250,1,7,250,250,1,1,7,1,m$buf)
        CALL f$pack2(zq,4,0,250,1,7,250,250,1,1,7,1,m$buf(29))
        CALL f$pack2(zp,4,0,250,1,7,250,250,1,1,7,1,m$buf(57))
        CALL MPI_SEND(m$buf,84,MPI_BYTE,m$to(1,0),0,MPI_COMM_WORLD,m$st
     1  atus)
      END IF
      IF (my$p(1) .GE. 1) THEN
        CALL MPI_RECV(m$buf,84,MPI_BYTE,m$to( -1,0),0,MPI_COMM_WORLD,m$
     1  retval,m$status)
        CALL f$unpack2(zm,4,0,250,1,7,0,0,1,1,7,1,m$buf)
        CALL f$unpack2(zq,4,0,250,1,7,0,0,1,1,7,1,m$buf(29))
        CALL f$unpack2(zp,4,0,250,1,7,0,0,1,1,7,1,m$buf(57))
      END IF
      DO l = 1,100
        t = 0.0037
        s = 0.0041
        IF (my$p(1) .LE. 2) THEN
          CALL f$pack2(zr,4,0,251,1,7,250,250,1,1,7,1,m$buf)
          CALL MPI_SEND(m$buf,28,MPI_BYTE,m$to(1,0),1,MPI_COMM_WORLD,m$
     1    status)
        END IF
        IF (my$p(1) .GE. 1) THEN
          CALL MPI_RECV(m$buf,28,MPI_BYTE,m$to( -1,0),1,MPI_COMM_WORLD,
     1    m$retval,m$status)
          CALL f$unpack2(zr,4,0,251,1,7,0,0,1,1,7,1,m$buf)
        END IF
        DO idx_1 = 2,6
          DO idx_2 = (MAX(( -250) * my$p(1) + 2,1)),(MIN(( -250) * my$p
     1    (1) + 999,250))
            za(idx_2,idx_1) = (zp(idx_2 - 1,idx_1 + 1) + zq(idx_2 - 1,i
     1      dx_1 + 1) - zp(idx_2 - 1,idx_1) - zq(idx_2 - 1,idx_1)) * (z
```

```
     2    r(idx_2,idx_1) + zr(idx_2 - 1,idx_1)) / (zm(idx_2 - 1,idx_1
     3    ) + zm(idx_2 - 1,idx_1 + 1))
          zb(idx_2,idx_1) = (zp(idx_2 - 1,idx_1) + zq(idx_2 - 1,idx_1
     1    ) - zp(idx_2,idx_1) - zq(idx_2,idx_1) * (zr(idx_2,idx_1) +
     2    zr(idx_2,idx_1 - 1)) / zm(idx_2,idx_1) + zm(idx_2 - 1,idx_1
     3    ))
        END DO
      END DO
      IF (my$p(1) .GE. 1) THEN
        CALL f$pack2(zr,4,0,251,1,7,1,1,1,1,7,1,m$buf)
        CALL f$pack2(zz,4,0,251,1,7,1,1,1,1,7,1,m$buf(29))
        CALL MPI_SEND(m$buf,56,MPI_BYTE,m$to( -1,0),2,MPI_COMM_WORLD,
     1  m$status)
      END IF
      IF (my$p(1) .LE. 2) THEN
        CALL MPI_RECV(m$buf,56,MPI_BYTE,m$to(1,0),2,MPI_COMM_WORLD,m$
     1  retval,m$status)
        CALL f$unpack2(zr,4,0,251,1,7,251,251,1,1,7,1,m$buf)
        CALL f$unpack2(zz,4,0,251,1,7,251,251,1,1,7,1,m$buf(29))
      END IF
      IF (my$p(1) .LE. 2) THEN
        CALL f$pack2(zr,4,0,251,1,7,250,250,1,1,7,1,m$buf)
        CALL f$pack2(za,4,0,250,1,7,250,250,1,1,7,1,m$buf(29))
        CALL f$pack2(zz,4,0,251,1,7,250,250,1,1,7,1,m$buf(57))
        CALL MPI_SEND(m$buf,84,MPI_BYTE,m$to(1,0),3,MPI_COMM_WORLD,m$
     1  status)
      END IF
      IF (my$p(1) .GE. 1) THEN
        CALL MPI_RECV(m$buf,84,MPI_BYTE,m$to( -1,0),3,MPI_COMM_WORLD,
     1  m$retval,m$status)
        CALL f$unpack2(zr,4,0,251,1,7,0,0,1,1,7,1,m$buf)
        CALL f$unpack2(za,4,0,250,1,7,0,0,1,1,7,1,m$buf(29))
        CALL f$unpack2(zz,4,0,251,1,7,0,0,1,1,7,1,m$buf(57))
      END IF
      DO idx_3 = 2,6
        DO idx_4 = (MAX(( -250) * my$p(1) + 2,1)),(MIN(( -250) * my$p
     1  (1) + 999,250))
          zu(idx_4,idx_3) = zu(idx_4,idx_3) + s * (za(idx_4,idx_3) *
     1    (zz(idx_4,idx_3) - zz(idx_4 + 1,idx_3)) - za(idx_4 - 1,idx_
     2    3) * (zz(idx_4,idx_3) - zz(idx_4 - 1,idx_3)) - zb(idx_4,idx
     3    _3) * (zz(idx_4,idx_3) - zz(idx_4,idx_3 - 1)) + zb(idx_4,id
     4    x_3 + 1) * (zz(idx_4,idx_3) - zz(idx_4,idx_3 + 1)))
          zv(idx_4,idx_3) = zv(idx_4,idx_3) + s * (za(idx_4,idx_3) *
     1    (zr(idx_4,idx_3) - zr(idx_4 + 1,idx_3)) - za(idx_4 - 1,idx_
     2    3) * (zr(idx_4,idx_3) - zr(idx_4 - 1,idx_3)) - zb(idx_4,idx
```

```
     3     _3 + 1) * (zr(idx_4,idx_3) - zr(idx_4,idx_3 - 1)) + zb(idx_
     4     4,idx_3 + 1) * (zr(idx_4,idx_3) - zr(idx_4,idx_3 + 1)))
          END DO
        END DO
        DO idx_5 = 2,6
          DO idx_6 = (MAX(( -250) * my$p(1) + 2,1)),(MIN(( -250) * my$p
     1    (1) + 999,250))
            zr(idx_6,idx_5) = zr(idx_6,idx_5) + t * zu(idx_6,idx_5)
            zz(idx_6,idx_5) = zz(idx_6,idx_5) + t * zv(idx_6,idx_5)
          END DO
        END DO
      END DO
      t$data = MPI_WTIME() - t$data
      PRINT *, my$proc, t$data
      CALL MPI_FINALIZE(m$status)
      END
```

REFERENCES

[1] P. Banerjee et al., "An overview of the PARADIGM compiler for distributed-memory multicomputers", *IEEE Computer*, 1995, To appear.

[2] E. Su et al., "Advanced compilation techniques in the PARADIGM compiler for distributed memory multicomputers", in *Proceedings of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995, pp. 424–433.

[3] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A data structure for compiling for distributed-memory multicomputers", in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, Montréal, Canada, Aug. 1994, pp. 123–132.

[4] D. J. Palermo et al., "Compiler optimizations for distributed memory multicomputers used in the PARADIGM compiler", in *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. II:1–10.

[5] P. Banerjee et al., "The PARADIGM compiler for distributed-memory message passing multicomputers", in *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, Dec. 1994, pp. 322–330.

[6] D. J. Palermo and P. Banerjee, "Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers", in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, Springer-Verlag, To appear.

[7] A. Lain and P. Banerjee, "Exploiting spatial regularity in irregular iterative applications", in *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995, pp. 820–827.

[8] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A convex programming approach for exploiting data and functional parallelism on distributed memory multicomputers", in *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. II:116–125.

[9] J. G. Holm, A. Lain, and P. Banerjee, "Compilation of scientific programs into multithreaded and message driven computation", in *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 518–525.

[10] C. D. Polychronopoulos et al., "Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors", in *Proceedings of the 18th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1989, pp. II:39–48.

[11] C. Koelbel et al., *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA, 1994.

[12] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.1", Tech. Rep., Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1994.

[13] W. S. Brainerd, C. H. Goldberg, and J. C. Adams, *Programmer's Guide to Fortran 90*, McGraw Hill, New York, 1990.

[14] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD distributed memory machines", *Communications of the ACM*, vol. 35, no. 8, pp. 66–80, Aug. 1992.

[15] Z. Bozkus et al., "Compiling fortran 90D/HPF for distributed memory MIMD computers", *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 15–26, Apr. 1994.

[16] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran", *Scientific Programming*, vol. 1, no. 1, pp. 31–50, Aug. 1992.

[17] Thinking Machines Corporation, Cambridge, MA, *CM Fortran Reference Manual, version 2.1*, 1994.

[18] S. Midkiff, "Local iteration set computation for block-cyclic distributions", in *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, Aug. 1995, pp. II:77–84.

[19] Applied Parallel Research, Placerville, California, *XHPF User's Guide, version 2.0*, 1995.

[20] The Portland Group, Inc., Wilsonville, Oregon, *PGHPF User's Guide*, 1995.

[21] Digital Equipment Corporation, Maynard, Massachusetts, *Digital High Performance Fortran 90 HPF and PSE Manual*, 1995.

[22] Vern Paxson, *Flex: A Fast Scanner Generator*, Free Software Foundation, Cambridge, MA, 1994.

[23] Charles Donnelly and Richard Stallman, *Bison: The YACC-compatible Parser Generator*, Free Software Foundation, Cambridge, MA, 1994.

[24] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley Publishing, Reading, MA, 1995.

[25] Message-Passing Interface Forum, "MPI: A message-passing interface standard", Tech. Rep. CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[26] F. McMahon, "The Livermore Fortran Kernels: A computer test of the numerical performance range", Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, Dec. 1986.