Optimizing Fortran 90D Programs for SIMD Execution

Gerald Roth

CRPC-TR93341-S
April 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Optimizing Fortran 90D Programs for SIMD Execution

Gerald Roth *

Thesis Proposal

Department of Computer Science
Rice University
Houston, Texas 77251

April 15, 1993

Abstract

SIMD architectures offer an alternative to MIMD architectures for obtaining high performance computation through parallelism. These architectures can offer impressive price/performance ratios for certain classes of problems. However, the effectiveness of such machines is greatly affected by the capabilities of the compilers which produce code for it. Current compilers have many weaknesses that introduce inefficiencies in the code that they produce. It is our thesis that advanced compiler techniques can produce more efficient SIMD code and exploit the massively parallel hardware closer to its full potential. To validate our thesis, we are designing and implementing compiler transformations that optimize computation and communication given the constraint of a single instruction stream.

1 Introduction

Parallel computing has been becoming more and more popular as a method of obtaining high performance. This trend will continue as parallel computers become less expensive and more readily available. However, programming such computers is still the major obstacle for their general acceptance. This is particularly true for parallel computers with distributed memory, since the programmer has to not only discover useful parallelism but also must distribute the data and the computation in a way such that communication costs do not overwhelm the benefits of parallelism.

To address the difficulties of programming distributed-memory parallel computers, Rice University has embarked on the Fortran D project to develop compilers that will address

*This work supported in part by the IBM Corporation through the Graduate Resident Study Program.

many of the impediments. Fortran D [25] combines the addition of annotations to the Fortran programming language with advanced compiler technology. The goal of the Fortran D project is to develop advanced compilers that take programs written in a *data-parallel* programming style and annotated with *data decomposition* statements, and automatically generate code that can be executed efficiently on different distributed-memory architectures.

The Fortran D source languages include both Fortran 77 [68] and Fortran 90 [5], and the target architectures may be either MIMD or SIMD. Several aspects of the Fortran D project are already being addressed. The following three projects are all currently underway:

1. The automatic annotation of Fortran 77 or Fortran 90 programs with data decomposition statements to produce the corresponding Fortran 77D or Fortran 90D programs [6, 44].

2. The generation of efficient MIMD code from Fortran 77D [30, 36, 64].

3. The generation of efficient MIMD code from Fortran 90D [20].

The scope of the work described in this document will be the generation of efficient code for distributed-memory SIMD machines from Fortran 90D. Compiling Fortran 77D to SIMD architectures will not be addressed in this work. However, using the techniques of automatic vectorization [3, 67] one can transform Fortran 77D into Fortran 90D, at which point the work described here can be employed to generate efficient SIMD code.

Distributed-memory MIMD architectures and distributed-memory SIMD architectures share many characteristics. Thus many of the optimizations used for compiling for MIMD machines are also useful for compiling for SIMD machines. However, the synchronous behavior of SIMD machines introduces many challenges not encountered in MIMD machines. A SIMD compiler must address these challenges, while taking advantage of the synchronous nature of execution in SIMD machines and the efficiencies that it affords.

It is our thesis that advanced compiler transformations can be used to optimize Fortran 90D programs for efficient execution on SIMD architectures, and in doing so exploit the massively parallel hardware close to its full potential.

The Fortran D project is designed to handle both regular and irregular computational patterns. This proposal addresses only Fortran D programs with regular computations. Irregular computational patterns are being addressed by others [31, 32, 33].

The remainder of this proposal is organized as follows. We first give a brief overview of the target SIMD architecture which this work assumes. Then in Section 2 we discuss related efforts. Section 3 describes our proposed research. We conclude in Section 4 with our research plan.

1.1 A Distributed-Memory SIMD Architecture

SIMD is an acronym for **S**ingle **I**nstruction stream, **M**ultiple **D**ata streams, and is one of the four categories proposed in Flynn's taxonomy of computer architectures [24]. A SIMD computer contains many data processors operating synchronously, each executing the same instruction from a single program counter. Each data processor is a fully functional ALU (Arithmetic Logical Unit). The SIMD architectures in which we are interested associate some local memory with each data processor. The data processor along with its associated

memory is referred to as a *processing element* (PE). The collection of all PEs is called the *PE array.* Each PE has an execution flag which can be set on or off to indicate whether the PE should execute the current instruction.

The PEs are connected by two interprocessor communication networks. These networks allow the PEs to access data that is stored in the memory of other processors. The first network is the global router. The global router allows the transfer of data between arbitrary pairs of processors. Quite often in the programs we are addressing, communication patterns are highly regular in that data is transferred between nearest neighbors. To support this type of communication efficiently, the second network forms a *NEWS* (*N*orth, *E*ast, *W*est, *S*outh) grid between the processors. We will assume that the NEWS grid is in the shape of a two dimensional nearest-neighbor mesh with wrap-around connections (*i.e.,* a *toroidal wrap* model).

Finally, there is the serial *front end* processor or *control unit.* The front end (FE) processor has two responsibilities. The first is to drive the PE array by broadcasting instructions and related data to all PEs. The second is to perform all scalar computations and control flow operations.

Our target architecture will not support virtual processors as is done on the CM-2 in Paris mode [21]. Instead, virtualization will be accomplished by the compiler by using *strip mining.* Like the MasPar Fortran compiler, we will know the size of the PE array at compile time, and we will use that knowledge to determine the strip amount.

For a broad discussion of general SIMD architectures see an appropriate computer architecture book [34, 38]. For a more complete description of the SIMD machines produced by Thinking Machines and MasPar see [35, 61] and [8, 54, 56] respectively.

2 Related Work

2.1 SIMD Distributed-Memory Compilers

2.1.1 Compass Compilers

Compass (1961-1991) was an independent software house which was involved in the design and implementation of several SIMD compilers. The front end and the global optimizer of both the CM Fortran and MasPar Fortran compilers were written by Compass; in fact, Compass wrote the entire initial CM Fortran compiler (Paris version). The CM Fortran and MasPar Fortran compilers are described in more detail below.

In addition to the general SIMD compiler development effort, Compass did research in the area of *data optimization* [1, 40, 41, 42, 43, 52]. The purpose of data optimization is to align data to improve locality and thus minimize interprocessor communication. Their method assumes an unlimited number of *virtual processors.* Then based on usage patterns, it maps arrays to the virtual processors, striving to align them so that communication costs are minimized. A later stage of the compiler then uses *strip mining* to map the virtual processors to the physical processors [65], also known as *array distribution.* This two stage approach makes each stage conceptually clean, but prevents them from interacting.

There are others, outside of Compass, who have done research in the area of data optimization. Chatterjee, Gilbert, Schreiber, and Teng [14, 15, 28, 29] describe algorithms to perform automatic array alignment at the statement level and the basic block level. Like

the Compass work, they separate the issues of array alignment from array distribution.

2.1.2 CM Fortran

Thinking Machines Corporation has developed two generations of a distributed-memory SIMD architecture, the most recent being the CM-2[1] [61]. CM Fortran, their Fortran derivative, is an implementation of Fortran 77 augmented with array constructs from Fortran 90. Their compiler for CM Fortran has also developed through two generations. The first generation was the *Paris* (or *fieldwise*) compiler which uses the bit-serial processors on the CM-2. The current CM Fortran compiler is the *slicewise* compiler. The slicewise compiler ignores the bit-serial processors and uses only the floating-point accelerator chips. The CM Fortran compiler can take advantage of the slicewise model of the machine in different ways to improve program performance for many engineering and scientific applications [58]. In this document, any references to the CM Fortran compiler will be to the slicewise compiler unless noted otherwise.

Even though the slicewise compiler gives improved performance, it also has several weaknesses. The compiler's shortcomings include the lack of transformations to increase the size of elemental code blocks, inefficient use of memory for compiler temporary arrays, and generation of poor code for communication along serial dimensions. Thinking Machines has documented many of the shortcomings [59], and has suggested methods that programmers may use to work around them.

Thinking Machines has also done some extensive work on compiling stencils [9]. A *stencil* is a computational pattern that calculates a new value for a matrix element by combining elements from neighboring matrix locations. The proper handling of stencils is very important to SIMD compilers, and can result in substantial performance gains. The performance gains are obtained by using multiwire NEWS communication, eliminating memory-to-memory copying of data, and full exploitation of the floating-point registers. In the current implementation of the stencil compiler, it is the responsibility of the programmer to identify a stencil computation, isolate it into a separate subroutine, and compile it with a special compiler.

2.1.3 MasPar Fortran

MasPar Computer Corporation has also developed two generations of a SIMD distributed-memory architecture: the MP-1 and the MP-2. Since both are architecturally equivalent from the compiler's point of view, we will only discuss the MP-2 [54]. Since Compass wrote the front-end for both the MasPar Fortran compiler and the CM Fortran compiler, it is no surprise that the languages the two compilers accept are quite similar.

A significant difference between the MasPar Fortran compiler and the CM Fortran compiler is how they handle data distribution. The MasPar Fortran compiler determines data distribution completely at compile time, whereas the CM Fortran compiler delays some distribution decisions until run time. Delaying these decisions until run time enables a single executable to run on machines with different number of processors. The cost of this flexibility is that the compiler cannot use the knowledge of the number of processors in performing optimizations.

---

[1] References to the CM-2 are meant to include the CM-200 [63] series.

2.1.4 Fortran-90-Y

Yale University has recently embarked on a compiler project for Fortran 90 [17]. The Fortran-90-Y compiler uses an abstract semantic algebra, *Yale Intermediate Representation* (YR), as its intermediate language. YR defines a series of semantic domains and sets of operators within each domain, and combines them with *shapes* that represent iteration spaces. The compiler optimizes a program by performing a sequence of source-to-source transformations over the YR code. Currently they produce code for the CM-2 by translating YR code into PEAC code, the same assembly language the CM Fortran compiler produces. Like the CM compiler and the MasPar compiler, the Yale compiler generates purely element-wise computations, *i.e.,* all operands must be perfectly aligned; a restriction we will eliminate. In addition to considering the data optimization problem [19], researchers on the Fortran-90-Y project have also looked into loop transformations that help optimize the node level programs when a `BLOCK` distribution is used [18].

2.1.5 MIMD Emulators

There has been growing interest in determining if SIMD architectures can successfully be used to handle problems that do not fit the data-parallel model [22, 37, 60, 66]. One method is to emulate MIMD execution by writing a SIMD program to interpret a MIMD instruction set. Each PE contains a program to be interpreted and its data. The front end then executes an interpreter loop where the instruction decode step iterates over all possible instruction types enabling only those PEs which have that instruction type as their next one to execute. The execution of that instruction is then simulated on the enabled PEs. This method is not particularly efficient though it does use the SIMD architecture in an interesting manner.

2.2 MIMD Distributed-Memory Compilers

The amount of research addressing compilation issues for distributed-memory MIMD machines is too voluminous to cover in depth in this paper. However, we will mention several related projects. All projects mentioned here exploit data-parallelism by using a *Single Program Multiple Data* (SPMD) model, in which the compiler generates a single node program which is executed on all the processors.

2.2.1 Fortran D

The effort at Rice University to compile Fortran 77D to MIMD machines has resulted in many advanced compiler analysis techniques and optimizations [30, 36, 64]. Many of the analysis techniques (*e.g.,* reaching decompositions) will also be very important for a SIMD compiler. However, many of the optimization techniques take advantage of the asynchronous nature of the processors and cannot be used on SIMD machines.

2.2.2 Crystal

The Crystal project at Yale University [16, 48, 50, 51] researched the issues of compiling a high-level functional language for SPMD execution. They made major contributions to the issues of automatic data alignment and identification of collective communication primitives.

2.2.3 Vienna Fortran

The Vienna Fortran project at the University of Vienna [7, 13, 70] is very similar, in both goals and methodologies, to the Fortran D project. The Fortran language is extended to allow the user to specify alignments and distributions, and the compiler automatically generates a message-passing SPMD program.

3 Proposed Research

Given a Fortran 90D program where all the parallelism is explicit, we propose a Fortran 90D SIMD compiler that will compile it for the target distributed-memory SIMD machine. During this compilation, several optimizing transformations will be performed. The main optimizations performed by the compiler deal mostly with communication issues. The importance of such optimizations cannot be overstated. They are much more important for SIMD machines as compared to MIMD machines, since all processors in the PE array must step through the communication instruction sequence even if they are not involved in the communication (in which case they ignore the instructions).

The Fortran 90D compiler will start by performing data optimization and identifying opportunities for collective communication. Stencil analysis and optimization will be performed next. That will be followed by context optimization. Finally, several phases will be performed that optimize communication operations by exploiting multichannel communication and handling offset arrays. Each of these phases is explained in turn.

3.1 Data Optimization

Two major tasks of a compiler for a distributed-memory machine are to distribute the data across the distributed memory and to assign computations to the processors. Fortran D directives enable a programmer to specify how the data is to be laid out across the memory of the machine. Compilers then often use the *owner computes* rule [12, 57, 69] to determine which processors will perform which computations. The owner computes rule states that computations are performed on the processors which will store the result of the computation.

Although the data distribution directives and the owner computes rule greatly simplify the distributed-memory compiler's task of determining where operations are to be evaluated, they can result in less than optimal code [42, 39]. Often it would be more efficient to compute intermediate results on processors other than those that are the target of the final results, or to change the processors on which the final results are stored. Issues include where operands reside with respect to each other, the cost of communication, and the amount of data to be moved. The task of determining where data should reside and where operations should be performed is known as *data optimization.*

In a Fortran D program, the programmer specifies a desired layout and alignment of all user declared arrays. Our proposed data optimizer will use the programmer's declarations as a starting point, and it will only be constrained by the fact that arrays are required to be in their declared positions at procedure entry and exit. Other than at the entry and exit of a procedure, arrays may be located such that communication costs are reduced. The data optimizer will also introduce temporary arrays as needed to allow intermediate results to be computed on different processors so as to decrease communication costs. We will also employ interprocedural analysis to determine which arrays need not be in their declared locations

around CALL sites. An array does not need to be in its declared location if it is not referenced or modified in the called procedure.

Our data optimizer will also handle explicit communication primitives that are available in Fortran 90. Our analysis will be able to examine intrinsics such as EOSHIFT, CSHIFT, and TRANSPOSE. When possible (and profitable), the optimizer will align the target array with the source array, thus eliminating the need for communication.

The structure of our data optimizer has not yet been finalized. We are thus not able to make a preliminary comparison against existing data optimization techniques. We are currently intending to use an annotated program dependence graph (PDG) [23] rather than using a preference graph as described by Knobe, Lukas, and Steele [41]. We believe annotating the PDG with regular section descriptors [10] will express the required array relationships in such a way to ease the analysis and produce alignments that require less communication. Not only will this framework ease the analysis required for data optimization, but it will also indicate locations to place the residual code motion, something that is currently done separately.

3.2 Collective Communication

After data optimization has been performed, the Fortran 90D SIMD compiler will generate explicit communication statements. It is desirable to make communication explicit as soon as possible so that it may be optimized by subsequent phases. This is the earliest point at which it is feasible to make communication explicit, since data optimization will affect what communication is necessary.

The research proposed in this phase consists of recognizing opportunities to exploit system supplied collective communication primitives that are optimized for the machine architecture to perform any communication that is required by the program. In particular, NEWS communication (*i.e.,* CSHIFT and EOSHIFT) will be used whenever possible. As stated in the beginning of this section, the exploitation of such system primitives is particularly important for SIMD machines, since SIMD machines cannot hide the communication latency by overlapping communication with computation.

For example, Figure 1 shows two versions of a short section of code. Assuming that data optimization has not altered the alignment specified by the programmer, this phase will recognize the need for communication, make the communication explicit, and isolate it in a separate statement. Notice how the second statement now contains only perfectly-aligned, element-wise array computations.

This work will be similar to that of Li and Chen [46, 47, 49, 50]. But whereas their work was based upon pattern matching of reference patterns, ours will be based upon pattern matching of dependence patterns within the annotated program dependence graph. It is our belief that this will ease the recognition process and allow us to match complex patterns without the need to break them into simpler components.

The user program may already contain calls to communication intrinsics, many of which may be embedded in expressions. We will take this opportunity to extract such calls from the expressions and place them in a separate assignment statement which defines a compiler temporary variable. The best alignment and distribution of the temporary will be determined by the compiler. The temporary variable will then be used in the original expression. This separation is necessary to prepare for subsequent phases which will attempt to optimize the

```
Before:  REAL, ARRAY(500) ::  A, B, C, D
         DECOMPOSITION DEC(500)
         ALIGN A, B, C, D WITH DEC
         ....
         A(1:100) = B(1:100) * C(2:101) * D(1:100)

After:   REAL, ARRAY(500) ::  A, B, C, D, TMP
         DECOMPOSITION DEC(500)
         ALIGN A, B, C, D, TMP WITH DEC       !TMP is aligned with the other arrays
         ....
         TMP = EOSHIFT(C,1,1)                 !TMP(1:100) will receive C(2:101)
         A(1:100) = B(1:100) * TMP(1:100) * D(1:100)
```

Figure 1 Collective Communication Insertion

communication primitives.

3.3 Stencils

Many engineering and scientific applications have computational kernels that can be classified as stencil computations. A stencil computation calculates a new value for a matrix element by combining elements from neighboring matrix locations. The elements are often combined by calculating the sum of products. Figure 2 shows a code segment performing a stencil computation that requires neighboring elements from the north, east, west and south.

The recognition and proper handling of stencils is very important to SIMD compilers, and can result in substantial performance gains. The performance gains are obtained by using multiwire NEWS communication, eliminating memory-to-memory copies and making excellent use of floating-point registers. The importance of stencils cannot be overstated; they occur as the computational kernels in many scientific and engineering applications.

Bromley *et al.* at Thinking Machines Corporation have already done some impressive work on compiling stencils [9]. They have developed a compiler called the convolution compiler, however most people simply refer to it as the stencil compiler. There are, however, still several opportunities for improvement. First of all, the task of identifying stencils rests on the shoulders of the programmer. The programmer must recognize the stencil computation, separate it out into its own subroutine and compile it with a special compiler. We proposed to develop a general recognizer for stencils. With the development of such a recognizer, stencil compilation will simply become a component of the general SIMD compiler that is invoked whenever a stencil computation is encountered. To incorporate recognition of stencils as part

```
REAL, ARRAY(1000,1000) ::  DST, SRC
DST = C1 * CSHIFT(SRC, 1, -1)
    + C2 * CSHIFT(SRC, 2, -1)
    + C3 * SRC
    + C4 * CSHIFT(SRC, 2, +1)
    + C5 * CSHIFT(SRC, 1, +1)
```

Figure 2 A Five-Point Stencil Pattern

of a general compiler, it must be robust enough to identify a stencil that may be spread out over several statements and interspersed with other statements. For this reason a recognizer would probably not work well at the source level or on an abstract syntax tree (AST). The recognizer we propose to build will work on the same annotated program dependence graph that is used during data optimization.

Our second improvement will address the stencil compiler's inability to handle large stencils. The current stencil compiler cannot be directly used on larger stencils since the register requirements exceed the machine's resources. We will address the issue of machine resource management in stencil code to alleviate this problem. This will be accomplished by weighing the cost of spilling registers used in the computation of the stencil against the cost of performing multiple substencil computations and combining the results. Determining the cost of computing multiple substencils is complicated by the fact that one must not only find an optimal partition of the stencil into substencils but must determine where the substencils are to be computed.

3.4 Context Optimization

Recall that in a SIMD architecture there is a dichotomy between the front end processor and the PE array. They are physically separate entities. Due to this, a SIMD compiler must generate two separate code streams: one for the front end and one for the PE array. The code for the front end makes calls to routines in the PE array code stream. These routines are called PECODE blocks.

All low-level optimizations, such as register allocation and instruction scheduling, are performed on each PECODE block separately. However, PECODE blocks tend to be quite small, thus limiting the usefulness of such optimizations. There are several factors that cause PECODE blocks to be small:

- **Context changes**: The code within a PECODE block executes elemental operations only on arrays which are conformable. Conformable arrays have the same shape and layout, or simply the same context. If statements operate on nonconformable arrays, the statements will be placed in separate PECODE blocks. For example, in Figure 3 arrays `A`, `B`, and `C` are one dimensional and arrays `X`, `Y`, and `Z` are two dimensional. Thus the three statements will cause three PECODE blocks to be generated.

- **Communication**: All communication operations are initiated by the front end. For this reason, a communication operation in the midst of array operations will cause separate PECODE blocks to be generated. See Figure 4 for an example.

- **Scalar code**: All scalar code is executed by the front end, and thus it necessarily causes surrounding parallel code to be placed in different PECODE blocks. Figure 5 contains an example.

It should be clear from the three examples that separate PECODE blocks can have a big effect on execution time. In each case, the use of the array `A` in the last statement cannot be made from a register and must access storage since register allocation does not span multiple PECODE blocks.

During execution the instructions in a PECODE block are broadcast to the PE array. These instructions may be executed within a loop if the size of the array being operated

```
REAL, ARRAY(1024) :: A, B, C
REAL, ARRAY(32,32) :: X, Y, Z
A = B * C                    ! generates three PECODE blocks
X = Y + Z                    ! due to different array contexts
B = A * 3.141
```

Figure 3

```
REAL, ARRAY(1024) :: A, B, C
A = B * C                    ! the CSHIFT will cause two
B = CSHIFT(C,1,1) + A        ! PECODE blocks to be generated
```

Figure 4

```
REAL, ARRAY(1024) :: A, B, C
REAL SCALAR1
A = B * C                    ! the assignment to SCALAR1 will
SCALAR1 = SCALAR1 + 1        ! cause the array operations to be
B = SIN(A) + SCALAR1         ! in separate PECODE blocks
```

Figure 5

upon is larger than the size of the PE array, in which case each PE handles multiple array elements. This loop is called the subgrid loop, as it iterates over the subgrid of local data that each PE has from the original array.

If the size of an array is not the multiple of the number of PEs, then when the array is distributed over the PEs some PEs will be given more data elements than others. Due to the single instruction stream, the number of iterations of a subgrid loop must be equal to the maximum number of elements that any one PE receives. When this occurs the PECODE block must include code to set the execution flags so that PEs do not execute the instructions currently being broadcast if they no longer have local data elements on which to operate. The code to set the execution flags is said to set the *context* of the PE array. This context setting code can account for a substantial amount of the execution time of a PECODE block.

3.4.1 High Level Context Optimization

To alleviate the problem caused by small PECODE blocks, we propose an optimization phase that will use a combination of dependence analysis and context analysis (considers array shape and layout) to perform code motion that will create separate blocks of scalar code, conformable parallel code, and communication code. The goal of this phase will be to maximize the size of PECODE blocks while maintaining the original semantics of the program. We call this optimization *context partitioning.*

As an example, the code in Figure 5 will be changed into that in Figure 6, for which a single PECODE block will be generated.

Cowie and Chen [17], in their work on the Fortran-90-Y compiler, state the goal of blocking computations over the same shape to form computational phases separated by communication phases. However, they currently only support a single transformation to address this goal: parallel loop fusion. Loop fusion is lacking in that it only considers adjacent loops.

The CM Fortran compiler performs this optimization, but only to a small extent [58].

```
REAL, ARRAY(1024) :: A, B, C
REAL SCALAR1
SCALAR1 = SCALAR1 + 1
A = B * C                    ! a single PECODE block will
B = SIN(A) + SCALAR1         ! be generated
```

Figure 6

The CM Fortran compiler also only considers adjacent loops, but it will attempt simple code motion to cause two loops to become adjacent. However, since it only performs minimal dependence analysis, the only code motion attempted is to move compiler-inserted scalar code if a larger parallel PECODE block will result.

Not only will this phase assist low-level optimizations, but by grouping communication code together it sets the stage for subsequent phases which optimize communication operations.

3.4.2 Low Level Context Optimization

There are optimizations which we may be able to perform at a much lower level. They attempt to optimize context changes that occur when some processors receive fewer elements of an array. Initially these optimizations consist of altering the code generation of strip-mined loops used for virtualization. We call this optimization *context splitting.*

To reduce the cost of context setting code (*i.e.,* code to set the execution flags on the PEs), we propose to alter the order that local elements are computed to reduce the number of context changes. The goal of the alteration is to process all elements that exist under one context before moving on to elements that exist under a different context. This alteration is achieved by modifying the strip-mined loops used for virtualization. By employing loop splitting and loop distribution we propose to isolate the iterations that require a context change. The code to set the PE context can then safely be hoisted from the strip mining loops, greatly reducing the number of times it is executed.

> EXAMPLE: Consider an array `B(11,11)` distributed on a 2 x 2 processor grid. Each processor will allocate a local array `B_LOCAL(6,6)`. All processors will have local data in elements `B_LOCAL(1:5,1:5)`. However, the two processors on the right side of the grid will not have data in their sixth column. Likewise, the two processors on the bottom of the grid will not have data in their sixth row. Note that the lower right processor is special as it is in the intersection of these two groups. Now when we process the statement `B = B + 1` the usual strip mining loops for virtualization will look like that in Figure 7.
>
> However, we know what the processor context is when processing the region `B_LOCAL(1:5,1:5)` — all processors have have data in these local elements. Similarly, we know the context when processing the regions `B_LOCAL(1:5,6)`, `B_LOCAL(6,1:5)`, and `B_LOCAL(6,6)`. We can take advantage of this knowledge by modifying the strip mining loops with a combination of loop splitting and loop distribution. Figure 8 shows the strip mining loops that eliminate redundant context changes. Note that this new code sets the context only four times

compared to the 36 times in the original code. □

The above example is simplified by the fact that each processor receives the same amount of data whether the data is distributed (BLOCK,BLOCK) or (CYCLIC,CYCLIC). In general however, a processor may receive different amounts of data for different distributions. We have algorithms to handle BLOCK, CYCLIC, and BLOCK_CYCLIC distributions.

3.5 Multichannel Communication

Some SIMD architectures (*i.e.,* the CM-2 in slicewise mode) have the capability to transfer data between two neighboring nodes in both directions simultaneously. This capability, known as *multiwire NEWS* communication, allows a node to send *and* receive data from each neighbor within the same communication operation. We propose to exploit this multiwire NEWS capabilities to reduce total communication costs.

Thinking Machines supplies a library routine (PSHIFT) which allows a programmer to use multiwire communications. One may combine several CSHIFTs and/or EOSHIFTs into a single PSHIFT. Such a combination is valid if the shifts are over different directions or axes and there are no dependences between their arguments. We also propose to investigate program transformations that will create opportunities for PSHIFT, such as breaking dependences between shift operations via use of compiler generated temporaries. Note that the analysis

```
X_EXTENT = (11 + (NXPROC-1)) / NXPROC        !  NXPROC = 2
Y_EXTENT = (11 + (NYPROC-1)) / NYPROC        !  NYPROC = 2
DO I=1, X_EXTENT
  DO J=1, Y_EXTENT
    set processor context based on I & J     !  set 36 times
    B_LOCAL(I,J) = B_LOCAL(I,J) + 1
  ENDDO
ENDDO
```

Figure 7 Naive strip mining loop

```
set processor context to turn on entire PE array
DO I=1,(X_EXTENT-1)
  DO J=1,(Y_EXTENT-1)
    B_LOCAL(I,J) = B_LOCAL(I,J) + 1
  ENDDO
ENDDO

set processor context to turn off right edge
DO I=1,(X_EXTENT-1)
  B_LOCAL(I,Y_EXTENT) = B_LOCAL(I,Y_EXTENT) + 1
ENDDO

set processor context to turn off bottom row
DO J=1,(Y_EXTENT-1)
  B_LOCAL(X_EXTENT,J) = B_LOCAL(X_EXTENT,J) + 1
ENDDO

set processor context to turn off right edge and bottom row
B_LOCAL(X_EXTENT,Y_EXTENT) = B_LOCAL(X_EXTENT,Y_EXTENT) + 1
```

Figure 8 Redundant context changes eliminated

required here is greatly simplified by the context analysis phase which has already grouped together much of the communication operations.

3.6 Offset Arrays

Shift operations (`CSHIFT`, `EOSHIFT`, and `PSHIFT`) occur frequently in Fortran 90 programs, either inserted by the programmer or by the compiler (see Sections 3.2 and 3.5). These operations take a source array, a shift amount and a dimension indicator as input. They then shift the source array the specified amount along the indicated dimension and store the result in the target array. For very large arrays distributed in a `BLOCK` fashion, memory-to-memory copying of the array within the local memories of the PEs dominates the execution time of such operations; up to 75% in some cases. But only in a few cases is the memory-to-memory copying actually necessary. Unfortunately, all the SIMD compilers mentioned in Section 2.1 make no attempt to determine if an entire copy of the source array must be made and thus they always incur the cost of this undesirable copying.

To alleviate this, we propose to perform compile-time analysis to determine whether an entire copy of the source array must be made for a call to a shift routine. If the analysis determines that an entire copy is not required, then we can treat the target array specially. We'll call such an array an *offset array.* Such arrays allow us to avoid the memory-to-memory copying; only the cross-processor data will be moved. We will exploit overlap areas [27] to store the communicated data locally within a PE. The use of overlaps will greatly simplify the task of generating code to drive the PE array when offset arrays are referenced. Program transformations, such as loop reversal, will also be used to allow more arrays to be treated as offset arrays.

This work has similarities to sectioning/strip mining of array syntax [2], vector register allocation [4], and scalarization of Fortran 90 code [67]. We hope to exploit this previous work in the analysis phase. The program transformations of the previous efforts will also be useful, but they will need to be augmented with distributed-memory specific transformations. Finally, the proper handling of offset arrays in Fortran 90 is useful for distributed-memory MIMD architectures as well.

4 Research Plan

This section describes our plans for implementing the Fortran 90D SIMD compiler, and a validation methodology to support our thesis.

4.1 Compiler Implementation

Our distributed-memory SIMD compiler will be implemented within the ParaScope programming environment [11]. It will be a source-to-source translator similar to other compilers/tools in ParaScope. Given a Fortran 90D program, the SIMD compiler will translate it into an equivalent program written in a subset of Fortran 90 suitable for a vendor's native Fortran 90 compiler. We will refer to this Fortran 90 subset as *Compass Fortran*, since Compass Incorporated wrote the front ends of both the CM Fortran and MasPar Fortran compilers. The output of the compiler may then be compiled by the Fortran compiler of the target SIMD machine [53, 62].

To support the analysis and optimization of Fortran 90 programs we will develop and

implement within our compiler algorithms for analyzing the usage of whole arrays and array sections. *Regular Section Descriptors* (RSD's) [10] will form the foundation on which these algorithms will be built. Data optimization, context optimization, and collective communication recognition will rely heavily upon the analysis performed by these routines.

Not only will this compiler support the validation of our thesis, but it will also establish the basis for future work on analyzing and optimizing Fortran 90 programs. This future work may include investigating compiler techniques for automatically generating SIMD code for *loosely synchronous* or wave-front style problems [26, 45] that is more efficient than the MIMD emulation methods described in Section 2.1.5.

4.2 Validation

To test the effectiveness of our optimizations, we will measure the improvements they produce above the system compiler on the target SIMD architectures, the CM-2 and the MP-1. The test cases we will use come from the Fortran D benchmark set [55]. We will time two versions of each Fortran 90D program in the benchmark set: the original Fortran 90 version and an optimized version that has been created by our SIMD compiler. The two versions will be compiled by the system compiler and their execution times compared. Even though this strategy will limit us to the capabilities of the system compiler, we should be able to show worthwhile improvements. At the Fortran 90 source level, we can perform data optimization, and high-level context optimization, as well as exploiting collective communication and multichannel communication.

Since low-level context optimization and the handling of offset arrays cannot be expressed at the Fortran 90 source level, we will not be able to test their usefulness with a source-to-source translator. To test these, we will be required to hand code them in a lower-level language (PEAC on the CM-2 or MPL on the MP-1).

As part of our validation work, we plan to perform an extensive investigation of data optimization. This investigation will evaluate the capabilities of different data optimization strategies, and validate their existence in SIMD compilers. The investigation will focus on the data optimizer we create as well as those introduced in Section 2.1.1. We will also study the relationship between data optimization and user directives such as the Fortran D `ALIGN` statement. We will investigate the use of data optimization in the following cases:

- the user supplies no alignment directives,
- the user supplies a set of naive directives for the major arrays, and
- the user supplies expert directives, complete with dynamic realignment where profitable.

5 Conclusion

Massively parallel SIMD machines offer an attractive method for obtaining significant performance improvements on parallel applications. Our thesis seeks to show that advanced compilation techniques can more fully exploit the SIMD hardware, thus increasing the usefulness of such machines. It should be noted that many of the optimizations proposed in this document are also applicable for compiling Fortran 90 for MIMD distributed-memory machines.

References

[1] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[2] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[3] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[4] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[5] ANSI X3J3/S8.115. Fortran 90, June 1990.

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[7] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[8] T. Blank. The MasPar MP-1 architecture. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[9] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[10] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[11] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[12] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[13] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[14] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, Xerox Corporation, December 1992.

[15] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on*

the Principles of Programming Languages, Charleston, SC, January 1993.

[16] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[17] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[18] M. Chen and Y. Hu. Optimizations for compiling iterative spatial loops to massively parallel machines. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[19] M. Chen and J. Wu. Optimizing FORTRAN-90 programs for data motion on massively parallel systems. Technical Report YALE/DCS/TR-882, Dept. of Computer Science, Yale University, December 1991.

[20] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C. Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[21] P. Christy. Virtual processors considered harmful. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 99–103, Portland, OR, April 1991.

[22] H. Dietz and W. Cohen. A control-parallel programming model implemented on SIMD hardware. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[23] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[24] M. Flynn. Very high-speed comnputing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[25] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[26] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[27] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice & Experience*, 2(3):171–193, September 1990.

[28] J. Gilbert and R. Schreiber. Optimal data placement for distributed memory architectures. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 462–471, Houston, TX, March 1991.

[29] J. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[30] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[31] R. v. Hanxleden. Compiler support for the machine independent parallelization of irregular problems. Thesis proposal, Dept. of Computer Science, Rice University, October 1992.

[32] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[33] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[34] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[35] W. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[36] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[37] P. Hudak and E. Mohr. Graphinators and the duality of SIMD and MIMD. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, July 1988.

[38] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, NY, 1984.

[39] K. Knobe, J. Lukas, and W. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[40] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers '88: The 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988.

[41] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[42] K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice & Experience*, to appear 1993. Special issue on Compilers and Programming Environments for Parallel Computers.

[43] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[44] U. Kremer. Automatic data alignment and distribution for distributed-memory machines in an interactive programming environment. Thesis proposal, Dept. of Computer Science, Rice University, March 1993.

[45] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[46] J. Li and M. Chen. Automating the coordination of interprocessor communication. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[47] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[48] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[49] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Dept. of Computer Science, Yale University, New Haven, CT, May 1990.

[50] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[51] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[52] J. Lukas and K. Knobe. Data optimization and its effect on communication costs in MIMD Fortran code. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[53] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, software version 1.1 edition, August 1991.

[54] MasPar Computer Corporation, Sunnyvale, CA. *MasPar System Overview*, March 1991.

[55] A. Mohamed, G. Fox, G. v. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, NPAC, Syracuse University, 1992.

[56] J. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proceedings of the 1990 Spring COMPCON*, San Francisco, CA, February 1990.

[57] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[58] G. Sabot. Optimized CM Fortran compiler for the Connection Machine computer. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, Kauai, HI, January 1992.

[59] G. Sabot, (with D. Gingold, and J. Marantz). CM Fortran optimization notes: Slicewise model. Technical Report TMC-184, Thinking Machines Corporation, March 1991.

[60] W. Shu and M. Wu. Solving dynamic and irregular problems on SIMD architectures with runtime support. submitted to ICPP'93, August 1993.

[61] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-2 Technical Summary*, April 1987.

[62] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 1.0 edition, February 1991.

[63] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-200 Tech-*

nical Summary, June 1991.

[64] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[65] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[66] P. Wilsey, D. Hensgen, N. Abu-Ghazaleh, C. Slusher, and D. Hollinden. The concurrent execution of non-communicating programs on SIMD processors. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[67] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[68] X3J3 Subcommittee. *American National Standard Programming Language Fortran (X3.9-1978).* American National Standards Institute, New York, NY, 1978.

[69] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[70] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran — a language specification, version 1.1. Interim Report 21, Institute for Computer Application in Science and Engineering, Hampton, VA, March 1992.