Compiler Technology for Machine-Independent Parallel Programming

Ken Kennedy

CRPC-TR93364
August 1993

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appeared in International Journal of Parallel Programming.

Compiler Technology for Machine-Independent Parallel Programming

Ken Kennedy

Center for Research on Parallel Computation

Abstract

Historically, the principal achievement of compiler technology has been to make it possible to program in a high-level, machine-independent style. The absence of compiler technology to provide such a style for parallel computers is the main reason these systems have not found widespread acceptance. This paper examines the prospects for machine-independent parallel programming, concentrating on Fortran D and High Performance Fortran, which support machine-independent expression of "data parallelism."

1.0 Introduction

One of the principal goals of compiler technology has been the support of a machine-independent programming interface. The substantial personnel resources required to develop a significant application program make it imperative that the programming investment be protected against the changes in target machine architecture that are a fixture of the evolving computer industry. Without this protection, the cost of application development would be prohibitive, because the entire application would need to be redeveloped every two to three years as newer parallel machines become available.

Compiler writers originally hoped that sufficiently powerful compilers would lead to languages expressive enough to multiply the productivity of the typical programmer. For a variety of reasons, this has not happened and the standard languages of today represent only a modest improvement over the original FORTRAN language. Fortunately, compiler technology has not been a complete flop. Although it was not originally developed for this purpose exclusively, the principal successes of compiler technology have been in the support of machine independence. In this paper we shall review those successes, and speculate on developments that are needed to extend this success into the realm of parallel machine architectures.

2.0 Scalar Optimizing Compilers

Until the advent of FORTRAN, computers were typically programmed in assembly language, which provided only limited programming support—mnemonic operation codes and symbolic addresses. The programmer was responsible for knowing the operation codes for the target machine and for building a program that efficiently executed on that target machine. Usually, the programmer had to start from scratch when a new target machine came along. The FORTRAN compiler permitted the programmer to specify programs in a more abstract notation that was closer to the notation used by scientists and engineers to describe calculations. Thus, programs could be written in a notation close to the one used in science and engineering. The main drawback of FORTRAN was that it concealed the details of the underlying target machine from the programmer, making it impossible for him or her to use the specialized tricks that assembly language programmers employed. It was the responsibility of the compiler to produce code that was efficient enough to be acceptable to the programmer. If the generated code was too inefficient, the user would abandon FORTRAN and the language would be a failure. In a retrospective article on the implementation of the original FORTRAN compiler, John Backus observed [1]:

> It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

Hence, the strong emphasis on "optimization," or the generation of efficient code, in FORTRAN compilers from the beginning. Because of this, the language was a huge success and programming now became possible for ordinary scientists and engineers. Assembly-language programming gradually died out and today is used only for small routines that must be highly optimized for a particular architecture.

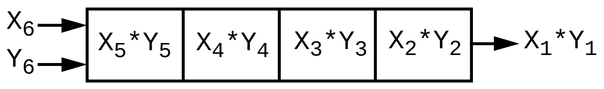
The emphasis on optimization has also been the source of machine independence in the language. Whenever a new scientific machine is introduced, a good optimizing compiler for FORTRAN is a must if that machine is to be a commercial success—all the machine-dependent tricks needed to achieve high efficiency are coded into that compiler. Hence, the programmer can concentrate on the desired computation, leaving machine-specific optimization to the compiler on each target machine. The ability to write machine-independent programs in FORTRAN is another reason for the success of the language.

Optimization technology for compilers continued to be refined throughout the sixties and seventies. One of the main tasks of an optimizing compiler was to do a good job of using *registers*, single-word memory locations in the processor. Data that can be kept in registers can be accessed more efficiently than data stored in the computer's main memory. The importance of register usage optimization increased dramatically with the introduction of so-called RISC (Reduced Instruction Set Computer) processors, which perform arithmetic operations only on registers. In fact, the entire RISC strategy was predicated on the assumption that all programming would be done in a compiled language, hence no effort need be given to economy of expression in the machine's instruction set. RISC machines have been very successful because their sophisticated compilers make it possible to move programs from conventional architectures with essentially no change and their simplified instruction sets permit much lower machine cycle times than had been possible with earlier single-chip systems. Thus, the user can realize a substantial performance improvement with little or no programming investment.

3.0 Vectorizing Compilers

Unfortunately, the transition to vector processors was not so easy. Vector processors were introduced in the mid 1970's to increase the computational power of high-end scientific processors, usually called supercomputers. It was discovered that, even though an arithmetic operation might take several machine cycles to complete, the arithmetic unit could produce a new result every cycle if it was pipelined—subdivided into stages such that each stage takes about a cycle and the intermediate results are captured at the end of each stage. Vector instructions were designed to keep pipelined arithmetic units busy by specifying that an operation be applied elementwise to two input vectors, or to one input vector and a scalar, to produce an output vector. In response to a vector operation, a pair of operands are introduced into the arithmetic unit on each machine cycle. When the pipeline is full, it is working on several pairs of operands at once, as depicted in Figure 1.

FIGURE 1. Pipelined Arithmetic Unit.



Although vector instructions can produce results at a rate many times that of a scalar computer, vector operations cannot be used in every computation. To understand why, consider the difference between the following two loops:

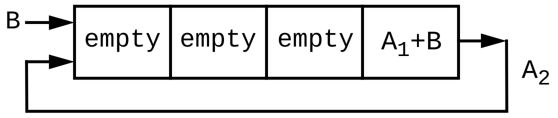
```
DO I = 1, N
   A(I) = A(I) + B
ENDDO
```

and

```
DO I = 1, N
   A(I+1) = A(I) + B
ENDDO
```

The first of the loops above can be executed efficiently on a pipelined arithmetic unit because each addition is independent of all the others, so each stage of the pipeline can be working on a different iteration of the computation. The second loop, however, presents problems because the output of each iteration is needed to compute the result in the next. Hence three stages of the pipeline must remain empty as depicted in Figure 2.

FIGURE 2. A Recurrence.



From this figure it is easy to see that the calculation of `A(2)+B` must wait for the calculation of `A(2)=A(1)+B` from the previous iteration to complete. Thus, a pipelined arithmetic unit can produce only one calculation every four cycles. This type of computation, in which the result on each iteration depends upon the result of some previous iteration, is called a *recurrence*.

The challenge of automatic vectorization is to distinguish between these two cases and generate vector code for the first. To accomplish this, David Kuck at the University of Illinois developed the theory of *dependence* in programs.

A statement S_2 is said to *depend upon* statement S_1 if there exists a path from S_1 to S_2 and S_2 uses a result computed by S_1. In the recurrence example above, the statement in the loop body *depends upon itself* because the loop provides a path to itself and it uses a result from its incarnation on the previous iteration. Kuck's basic vectorization principle can be paraphrased as follows [2]:

> **Vectorization Principle.** A statement in a loop can be directly vectorized with respect to that loop if that statement does not depend upon itself, either directly or indirectly.

Statements that cannot be directly vectorized may still be made to execute efficiently by employing various transformations that can expose full or partial parallelism. The theory of vectorization is rich with these transformations [3,4,5,6,7].

As a result of the work of Kuck and others, automatic vectorization technology has been incorporated into almost every FORTRAN 77 compiler for a vector machine. Does this mean that FORTRAN 77 programs written before the appearance of vector computers can be run efficiently on them simply by recompiling without changing the source? For the most part, the answer is *no*. Automatic vectorization did *not* solve the so-called "dusty deck problem." Almost every program run on a vector machine was rewritten to exploit its features. However, vectorization made it possible to rewrite those programs in a subdialect of FORTRAN 77, the "vectorizable loop subdialect," consisting of loops that vectorize well. Sophisticated vectorizing compilers ensure that programs written in this subdialect achieve high performance on almost all vector machines. The vectorizable loop subdialect, well documented in FORTRAN manuals for vector machines, consists of loops without recurrences. When recurrences are required, the use of special forms that the compiler recognizes can often achieve partial vectorization. In any case, the code can be run on vector or scalar computers without change and with performance on vector machines that is close to the best possible for the given algorithm.

The great success of vectorization technology was providing a machine-independent programming interface for vector machines. Once rewritten in the vectorizable loop subdialect of FOR-

TRAN, a program can be compiled and executed efficiently on any vector or scalar machine. This technology, which was developed between the mid-seventies and mid-eighties, was responsible for the enormous success achieved by vector processors during that period. Without a simple, machine-independent programming interface, these processors would not have been accessible to the majority of scientific users.

4.0 Parallelism: the New Challenge

In search of ever higher performance levels, the high-performance scientific computing community is turning to parallel systems with hundreds or even thousands of processors. Parallel architectures are appealing because of the revolutionary advances in the performance of commodity single-chip processors, such as the Intel i860, the MIPS R4000, the HP PA-RISC and the DEC Alpha. If a computer constructed by interconnecting a large number of these processors via a high-speed network can be used to perform a single scientific computation efficiently, then such a machine presents a cost-effective alternative to high-end vector computers, which are typically built with custom VLSI.

The only problem with this vision is the requirement that the parallel machine can be used *efficiently* on a single problem. Most parallel computer systems experience high overhead in starting and synchronizing a parallel computation. Therefore, for a parallel computation to be efficient, there must exist regions that can be executed in parallel with a coarse enough granularity to compensate for this overhead. Reconstructing applications to meet this requirement is one of the principal challenges of parallel programming.

But even if this challenge can be met, we are left with the question: How should parallel programs be expressed? This question is the subject of the remainder of the paper.

4.1 Parallel Computer Architectures

The problem of machine-independent parallel programming is compounded by the variety of different machine architectures available. Currently, four distinct types of parallel machines are available:

1. *Uniform-access, shared memory.* Machines in this class have a global shared memory fully addressable from each of the processors such that each every access to global memory takes

the same amount of time. Examples include multiprocessor workstations, such as those produced by Silicon Graphics, and multiprocessor vector supercomputers, such as those from Convex, Cray, NEC, Fujitsu, Hitachi and IBM. The typical implementation method for these systems is via a high-bandwidth bus attached to the processors and memory modules. Because of limitations in the performance of these busses, this type of architecture is generally regarded as non-scalable. Hence, future parallel supercomputers will use some other interconnection method.

2. *Asynchronous distributed memory.* In this class, each processor is packaged with its own memory and executes independently of the other processors, with synchronization handled through communication. Such processors are often called MIMD, for “multiple instruction, multiple data,” because they can execute different instructions on different data elements.All of the processors are interconnected via a scalable network such as a hypercube, mesh or “fat” tree. Thus a global data array must be partitioned across the memories of the different processors and, whenever a processor needs a datum that is not in its own memory, it must collaborate with the owner of that datum to receive its value via the communication network. Part of the motivation for this design is it permits a machine to use more memory than any single processor can address, an important consideration when the standard microprocessor address length is 32 bits. Examples of this architecture are the NCube 2, the Intel Paragon, the IBM SP-1 and the Thinking Machines CM-5. A major problem with this design is that software latencies are incurred on both endpoints whenever a message is sent. These latencies are so large that the most important optimization is to minimize the number of messages sent through good data placement and combining messages.
3. *Synchronous distributed memory.* Machines in this class, which are often called SIMD for “single-instruction multiple-data,” also have distributed memory but each processor executes the same instruction on each machine cycle. Thus, these machines are capable of exploiting parallelism at finer granularity than asynchronous machines but they suffer when presented with code that includes conditionals, because each side of the branch must be executed in sequence with different sets of processors turned on and off. Examples are the Thinking Machines CM-2, the ICL DAP and the MasPar MP-1.
4. *Non-uniform-access shared memory.* Machines in this class have a shared memory in that each processor can access all of the machine’s memory. However, the processors are each packaged with part of the global memory, so that accesses to local memory are fast while accesses to remote memory may be very slow. This design will become increasingly prominent as microprocessors with 64-bit addressing become the rule rather than the exception. On these machines, which currently include the Kendall Square KSR-1 and the BBN TC-2000, good data placement is again a key to high performance because it minimizes the number of expensive references to remote memories.

Each of these machines has advantages and disadvantages and there is no reason to expect that the range of architectures will narrow anytime in the near future.

4.2 Machine-Independent Parallel Programming

The programming systems available on current parallel machines are primitive in the sense that they reveal much of the architecture of the target machine. Distributed-memory machines offer message-passing libraries in Fortran and C. Shared-memory machines permit parallel loops to be specified. SIMD machines depend on variants of Fortran 90, with its multidimensional array assignment statements.

The difficulty with reliance on machine-dependent programming interfaces is that users cannot expect their investment in programming a parallel machine to be protected—when a new state-of-the-art parallel computer appears one or two years in the future, they may be required to completely reprogram the application to achieve high performance. This problem is dramatically illustrated by users' experience in moving from the Thinking Machines CM-2, a SIMD machine, to its successor, the asynchronous, distributed-memory CM-5. Programs in CM Fortran optimized for the CM-2 do not achieve high performance on the CM-5. Currently, the best way to get high performance on the CM-5 is to rewrite the program completely in message-passing Fortran [8].

The absence of a standard, machine-independent programming interface for parallel computer systems is the principal reason that these systems have seen so little success in the commercial marketplace. Sales of parallel computing systems are primarily to research labs—these systems are infrequently selected for the main-line applications upon which a company's business depends. Those applications remain targeted to vector supercomputers. A key constituency that must be sold for massively parallel systems to be a success are the third-party software vendors, like McNeil-Schwendler, Swanson Analysis, IMSL and NAG. These companies have been very reluctant to retarget to parallel machines because of the machine-dependence problem.

For these reasons, adoption of standard languages for machine-independent parallel programming is the key to the future success of parallel computation.

To avoid setting a goal that is too ambitious or not ambitious enough, let us establish what we mean by "machine-independent." We will say that a programming language and its associated compilers *support machine-independent parallel programming* if, for a given source program, the compiled code achieves performance on each target machine comparable to the performance of the best version of that same algorithm hand-coded in the standard machine-dependent programming interface for that target. In other words, we expect good performance when the algorithm is well suited to the target architecture and we expect it to run as well as possible on an architecture for which it is ill-suited.

4.3 Automatic Parallelization

One solution to the problem of machine-independent programming would be to repeat the success of vectorization by identifying a subdialect of Fortran 77 from which efficient parallel programs can be generated for a variety of target architectures. This would imply the development of a technology to automatically find the parallelism inherent in the source program and tailor that parallelism to the target machine.

The principal challenge that automatic methods must overcome is the granularity problem. That is, an automatic parallelizer must be able to find parallel regions of sufficient granularity to compensate for the overhead of parallel execution, particularly on asynchronous (MIMD) processors. Vectorization could concentrate on finding inner loop parallelism because of the fine-grain parallelism inherent in vector instructions. For parallel machines, on the other hand we must look for parallelism in large outer loops. Coarse-grain parallel loops are also likely to contain subroutine calls, as it would be overly burdensome to expect the user to eliminate all procedure calls by hand in order to run a program on a parallel machine.

Seeking coarse-grain parallelism is difficult because of the limitations of program analysis techniques. The theory of dependence again plays a central role in parallelization. A dependence is said to be *carried* by a loop if the source and sink of the dependence are on different iterations of that loop [3]. This definition is used in the following analog of the vectorization principle [9]:

> **Parallelization Principle.** A loop can be parallelized without inter-iteration synchronization only if the loop carries no dependence.

When parallelizing loops in an existing program, a compiler must be conservative. A parallelizing compiler tries to prove that a target loop can be parallelized by proving that no carried dependence exists. For each pair of subscripted array references that might give rise to a carried dependence, the compiler must show that it gives rise to no dependence carried by the target loop. Each such proof is subject to limitations in the program analysis system. For example, consider the loop:

```
DO I = 1, N
   CALL SOURCE(A, I)
   B(I) = A(I) + C
ENDDO
```

To parallelize this example, the compiler must be able to prove that subroutine `SOURCE` does not store into any location of array `A` except for `A(I)`. Analysis of side effects to array subsections is a complicated form of interprocedural analysis. Although researchers have developed methods for this kind of interprocedural analysis, these methods are limited in precision by a requirement that, in a practical compilation system, no analysis should take time that is nonlinear in the size of the whole program. Even within a single compilation unit, it can be difficult to carry out a precise analysis of dependences because many subscripts contain references to variables that are not known until run time.

By the parallelization principle, it takes only one failure of the dependence system to incorrectly prevent parallelization of an important loop. Thus, a parallelization system is more likely to fail on a large loop, particularly one that contains calls to subroutines and functions, because there are more opportunities for failure.

In addition to finding coarse-grain parallelism, the automatic system must tailor that parallelism to a specific target machine. If the target is a distributed-memory machine, the data arrays in the program must be distributed to the individual processors in a way that minimizes the communication required during execution of the program. This data distribution problem is widely viewed as the fundamental challenge to programming distributed memory machines [10]. Although a substantial amount of research has been conducted on automatic methods for selecting data distributions, this problem is still considered too hard for practical compilers.

For these reasons, the parallelization research community has generally concluded that automatic methods, by themselves are not going to be sufficient for the specification of parallel programs. This is not to say that automatic systems have yielded unimpressive results. It merely means that in many cases the programmer will need to provide additional instructions to the parallel programming system to handle interesting programs. In other words, the programmer will need some way to indicate explicitly where the parallelism is to be found.

The upshot of all this is that we must look to language extensions for parallel programming. Extensions of this sort are the subject of the remainder of the paper.

4.4 Extended Languages

We will examine potential language extensions for two kinds of parallelism: *data parallelism* and *task parallelism*. *Data parallelism* arises from partitioning the data domains of a program and using the data partition to imply a partition of computation. In data parallel applications, a computation is applied to every element of a data domain and parallelism is achieved by assigning each processor to a subdomain of the whole domain. For example, if every element of a 1000-element array is to be updated by a constant on a 10-processor parallel machine, each processor can be assigned to update 100 of the elements. Data parallelism is the most common form of parallelism in scientific calculations because it permits larger problems to be solved by increasing the number of processors

Task parallelism partitions the problem solution into functionally-different tasks, each of which represents a different part of the computation. Any tasks that do not perform computations on the same data may be executed in parallel. Although task parallelism cannot be used to keep an unbounded number of processors busy, it is nevertheless an important part of the overall parallelism picture in a given application.

As we shall see shortly, these two types of parallelism require substantially different linguistic approaches.

4.5 Data Parallelism

We begin with a discussion of possible approaches to machine-independent specification of data parallelism. One question that arises is whether one of the existing languages would be a good vehicle for this task. In this section, I discuss each of the existing models in turn.

4.5.1 Message-Passing Fortran

The most common vehicle for parallel programming on scalable parallel computers is some form of Fortran with message-passing primitives, which I shall henceforth refer to as *message-passing Fortran*. The most commonly-used programming style in message-passing Fortran is called *single-program, multiple-data* or *SPMD*. In this style, the programmer produces a general program for a single node of the processor array, which is intended to be executed on every node in the array. Global arrays must be explicitly distributed across the processors into local arrays on each node, and any access by one processor to data residing on another must be implemented by having the owning processor execute a “send” primitive with the required data as a parameter and having the accessing processor execute a “receive” primitive. In other words, the user must choose a data decomposition for the global arrays, strip mine all the loops to run on a single processor and produce explicit communication calls for all interprocessor communication. To achieve high performance on most problems, the user is also responsible reducing communication costs by blocking short messages into longer ones and overlapping communication with computation.

Although message-passing Fortran has the advantage of forcing the programmer to deal with the central intellectual problem of programming a distributed-memory system—the distribution of data to individual processor memories—most users find programming these systems to be incredibly tedious. Users would prefer to write in a language that provided a shared name space and they object to being required to manage all the details of communication, including optimization. Although there are a few exceptions, most users who have written one program in message-passing Fortran never want to write another. The only reason that they use this language to begin with is to get performance that is not otherwise possible on computer systems available to them. We need to find a method of programming that offers high performance without the pain of explicit message passing.

4.5.2 Parallel-Loop Fortran

Another parallel programming language that is commonly used for scientific applications is FORTRAN 77 extended to allow parallel loops and parallel case statements. Since the Parallel Computing Forum (PCF), an informal standardization group, proposed a standard for this style of extension, this language is often referred to as “PCF Fortran” [11].

Historically, parallel loop Fortrans were developed for uniform-access shared-memory parallel computer systems that used a bus to connect processors to global memory. Issues of data distribution are much less crucial on these machines, because each processor experiences a uniform access time to global memory. Thus the features for explicit specification of data placement are much weaker than those in message-passing Fortran. In particular, PCF Fortran only provides a mechanism for declaring data to be private to a particular processor. For this reason, it seems unlikely that this language can serve as a convenient vehicle for machine-independent data parallel programming, although some of its features might well be incorporated into such a language.

4.5.3 Fortran 90

A final candidate is Fortran 90, the standard language for programming SIMD processors like the CM-2 and MasPar MP-1. Fortran 90 provides multidimensional array assignment statements, which are ideal for specifying fine-grain, synchronous parallel operations. In addition, it is not too difficult to “strip mine” or subdivide these assignment statements to run on a coarse-grain asynchronous parallel processor, with each processor executing a block of the assignments specified. Thus, it seems to be an interesting candidate for machine-independent parallel programming. However, Fortran 90 provides no assistance for the data layout problem, a serious limitation. In fact, Thinking Machines has added explicit directives for data layout in CM Fortran to overcome this problem.

4.5.4 Fortran D

These considerations suggest another approach, which has been followed in Fortran D, a research language developed at the Center for Research on Parallel Computation. Fortran D is based on an idea that was developed over a number of years by the community of researchers in compilation for distributed-memory asynchronous machines [12,13,14,15,16]: Since data layout is the key intellectual problem in programming these machines, why not make it the basis for a parallel programming system?

To that end, Fortran D extends both FORTRAN 77 and Fortran 90 by the addition of three new statements for the specification of data distribution:

1. The `DECOMPOSITION` statement specifies a fine-grain virtual processor array with as many elements as the given problem could effectively use.
2. The `ALIGN` statement permits different data arrays to be aligned with a decomposition. For highest performance, the compiler should allocate any two array elements that are aligned to the same decomposition element to the same processor.
3. The `DISTRIBUTE` statement permits the specification, in a machine-independent fashion, of how decomposition elements should be mapped to the actual processors.

A powerful property of these decomposition statements is that they do not affect the meaning of the program being compiled. In other words, if there is enough memory, a Fortran D program can be correctly implemented by simply ignoring the Fortran D extensions. However, compilers for high-performance parallel computers can use the distribution information to compile good code by strip mining according to the data layout. An experimental compiler constructed at Rice University, along with a number of earlier experimental systems have demonstrated the feasibility of this approach, at least for “regular-mesh” problems in which all subscripted variables are simple linear expressions [17,18].

4.5.5 High Performance Fortran

The original technical report on Fortran D [19] was released in December of 1989 and received widespread attention. A number of manufacturers expressed interest in producing an informal standard for a language called High Performance Fortran (HPF) based in part on the notion of data distribution. At Supercomputing 91 in November 1991, interested parties met in a birds-of-a-feather session and agreed that the Center for Research on Parallel Computation would convene a standardization activity. The resulting group, called the High Performance Fortran Forum, was launched in January 1992 with the goal of producing an informal standard in one year. Meeting every six weeks, the working group was able to produce a draft for public comment by February 1993 and the final form of the language was agreed to in March 1993. The final HPF language definition is scheduled to be available May 15, 1993.

High Performance Fortran differs from Fortran D in two major ways:

1. HPF is based exclusively on Fortran 90, while Fortran D had bindings to both Fortran 77 and Fortran 90.
2. Fortran D required interprocedural propagation of distributions, while HPF has carefully avoided such a requirement in the interest of practicality.

It remains to be seen whether the product compilers expected in late 1993 (so far, Thinking Machines, Intel, DEC, and MasPar have announced that they will deliver products) will make High Performance Fortran efficient enough to be practical for typical scientific applications.

4.5.6 Irregular Problem Support

In the interests of time, the HPF Forum decided to leave many proposed features out of the language. The most important of these, in my opinion, is support for “irregular problems.” Irregular problems are those simulation problems defined on irregular or adaptive meshes. They include finite element methods and sparse linear algebra. From the compiler-writers viewpoint, an irregular problem can be characterized as one in which subscripted index array references appear in data array subscripts.

Fox estimates that at least fifty percent of all scientific programs are irregular and that more than fifty percent of all supercomputer cycles are spent on these programs. Therefore, it is imperative that we be able to effectively support the parallelization of irregular problems.

The most successful approach to handling irregular problems is the one pioneered by Joel Saltz and colleagues at ICASE and Purdue University [13,20,21]. To parallelize an irregular problem, the programmer must begin with a load-balancing or "partitioning" phase, in which problem elements are mapped to processors. Then the programmer must generate all communications that are implied by the resulting processor assignment. Saltz's approach is to subdivide the latter task into an *inspector*, which determines the communication patterns that are required and sets up data structures and schedules to carry them out efficiently, and an *executor*, which carries out the communications on each program step. If the irregular mesh changes slowly or not at all, the load balancing and inspector will be required infrequently, making it possible to amortize the cost over many time steps. Saltz's group has developed an extensive library of run-time primitives to automate many of the more tedious details of the implementation.

One question that naturally arises is: Can the compiler do most of the work in generating inspector-executor style communications? Saltz believes that it can and has embarked on a project with my group at Rice and Geoffrey Fox's group at Syracuse to provide such support in the Fortran D compilers.

The simplest form of support proposed is based on a special form of the `DISTRIBUTE` statement available in Fortran D:

```
            DISTRIBUTE D(MAP)
```

where `MAP` is an array or function that is constructed at compile time by the load balancer and maps each element of a decomposition to the processor to which it is assigned. If the programmer provides a partitioner to compute `MAP`, the Fortran D compiler will handle all the details of constructing and positioning the inspectors and executors, using calls to the Saltz runtime library. The programmer need not make any other explicit changes to the code.

Currently, an effort is underway to validate these techniques in the Fortran D compiler and to extend them to include various ways for automatically generating partitioners from architecture-independent hints provided by the programmer. If the compiler technology is successfully demonstrated, it will be a candidate for inclusion in a second round of HPF standardization scheduled for calendar year 1994.

4.6 Task Parallelism

Explicit programming approaches to task parallelism have benefitted from extensive experience with this kind of parallelism in the computer science systems community, and most of the effective approaches have arisen from that community. Machine independence for task-parallel problems is achieved by constructing processes that are natural to the problem solution and letting the system assign these processes to real processors. The principle challenge has been to make it possible to produce tasks that are lightweight enough to achieve acceptable efficiency on the natural granularity of scientific problems. In most cases this has led to schedulers that operate in user space rather than kernel space.

There are however a number of interesting challenges for the programming systems developer. One such is how to make specification of tasks palatable to the typical scientific programmer who codes in Fortran. Task parallelism usually implies that the tasks have independent data spaces and all data sharing is achieved through explicit communication. Research is being conducted on a number of approaches to this problem, which I shall briefly survey:

1. In *Linda*, the user is able to share data through a global tuple space, which can be written to and read from any process via special primitives [22]. Process synchronization is also provided via read and write primitives, as one form of the read primitive blocks when a tuple with the specified key does not yet exist in the tuple space. When the desired tuple is written by another process, the blocked read primitive will be reactivated to return the new tuple.
2. *HeNCE*, developed by Dongarra and his colleagues at the University of Tennessee and Oak Ridge National Laboratory, is an interactive graphical programming system that allows the user to specify synchronization and communication between subroutines in a Fortran or C program [23]. In the HeNCE graph, a node represents a process, which is an instantiation of some procedure, and each directed edge represents a communication or synchronization constraint. That is, the process at the sink of the edge cannot be scheduled until the process at the source has satisfied the edge's constraint by completing execution and communicating the

required variables. The HeNCE system then generates a C program with calls to PVM (Parallel Virtual Machine), a message passing library [24], for synchronization and communication.

3. *CODE* (computationally-oriented display environment), developed by J.C. Browne and his group at the University of Texas is an interactive graphical programming system, similar to HeNCE. It provides a rich collection of communication types and it is also hierarchical, in the sense that nodes in a graph can represent a subgraph, expandable on demand. CODE can be used to generate programs in any of a variety of task-parallel programming languages [25].
4. *Fortran M* (for modular) and *Concurrent C++* (CC++) are extended languages developed by Mani Chandy's group in the Center for Research on Parallel Computation [26,27]. These languages provide typed message passing, so that it can be established at compile time that a program is deterministic. In addition, many other properties can be established by formal means. CC++ includes an additional innovation, the *sync* variable, which is an extension of the C++ typed constant. A *sync* variable is a single-assignment variable that can be used for synchronization. Any process that attempts to use a sync variable before it is defined will block until the definition is available. If all data sharing is via sync variables, a CC++ program is guaranteed to be deterministic. A collaborative project is underway to permit CC++ and Fortran M programs to be generated from the CODE system.
5. *Automatic discovery of task parallelism* is the subject of research by a number of investigators, most notably Fran Allen and her group at IBM Research [28,29] and Constantine Polychronopoulos at the University of Illinois [30,31,32,33]. Although this work shows great promise, it is still too early to tell if the compiler technology will be practical and I am aware of no commercial compiler that attempts an aggressive task parallelization.

Outside the automatic task parallelization community, not much work on compiler optimization for task parallelism has been done, since most of the other approaches rely almost exclusively on run-time libraries. There are two exceptions. Linda has a precompiler that can optimize calls to Linda primitives in context [34] and the Center for Research on Parallel Computation has begun a project to unify Fortran M and CC++ with Fortran D, making it possible to use the Fortran D compiler technology to optimize programs that intermix task and data parallelism. In addition, a few restructuring environments such as E/SP, a commercially-available system from SES that is based on Browne's work [35], have dealt with task parallelism.

5.0 Summary and Conclusions

Progress in compiler technology over the past few years has brought within reach a solution to the problem of writing machine-independent data-parallel programs. If successful, Fortran D and High Performance Fortran may make it possible to maintain a single image of a Fortran program for a variety of parallel machines. A research project led by Dennis Gannon at Indiana University is developing pC++, which addresses the same problem for C++ programs [36]. When combined with process-based methods for specifying task parallelism, these languages and their associated compiler technology may represent a first step toward a truly comprehensive solution to the portable parallel programming support.

Nevertheless, there is still much to do. Compiler optimization research is needed to make sure that HPF compilers generate code that is comparable to hand code in message-passing Fortran, HPF needs to be extended to handle task parallelism, and language and compiler technology for support of irregular calculation must be perfected and validated.

Finally, the Fortran D group is now beginning work on a project to build a programming environment for Fortran D and HPF that can help the user find bugs and eliminate performance bottlenecks in the generated programs. The key technical problem will be to present debug and performance information in terms of the HPF source code rather than some low-level object program that bears little resemblance to the original HPF.

If all of these efforts are successful we may see scalable parallel computers that are truly usable—usable enough to attract commercial applications developers. Only then can parallel computing realize its enormous promise.

6.0 References

[1] J. Backus. The history of FORTRAN I, II and III. *ACM SIGPLAN Notices* 13(8): 165–180, August 1978.

[2] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. *Proceedings of COMPSAC 80, The 4th International Computer Software and Applications Conference,* pages 709–715, Chicago, Illinois, October 1980.

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

[4] K. Kennedy, K. S. McKinley and C. Tseng. Analysis and transformation in the ParaScope Editor. *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[5] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe. Dependence graphs and compiler optimizations. *Conference Record of the Eighth Annual ACM Symposium on the Principals of Programming Languages*, pages 207–218, Williamsburg, Virginia, January 1981.

[6] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass., 1989.

[7] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, NY, 1991.

[8] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* Ph.D. Dissertation, Rice University, January 1993.

[9] J. R. Allen, D. Callahan and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, Munich, Germany, pages 63–76, January, 1987.

[10] 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.

[11] Parallel Computing Forum. PCF: parallel Fortran extensions. *Fortran Forum* 10(3), September, 1991.

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

[13] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed Systems* 2(4), October 1991.

[14] A. Rogers and K. Pingali. Process decomposition through locality of reference. *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[15] M. Rosing, R. Schnabel and R. Weaver. Expressing complex parallel algorithms in DINO. *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, California, March 1989.

[16] H. Zima, H.-J. Bast and M. Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization.

[17] S. Hiranandani, K. Kennedy and C. Tseng, Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. *Proceedings of the ACM 1992 International Conference on Supercomputing*, Washington, DC, pages 1-14, July 1992.

[18] 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.

[19] 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.

[20] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol and K. Crowley. Principles of runtime support for parallel processors. *Proceedings of the Second International Conference on Supercomputing,* St. Malo, France, July 1988.

[21] J. Wu, J. Saltz, S. Hiranandani and H. Berryman. Runtime compilation methods for multicomputers. *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, Illinois, August 1991.

[22] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, April 1989.

[23] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing.*Proceedings of Supercomputing 91,* pages 435-444, Albuquerque, New Mexico, November 1991.

[24] A. Beguelin, J. Dongarra, A. Geist, R. Manchek and V. Sunderam. Opening the Door to Heterogeneous Network Supercomputing. *Supercomputing Review* 4(9): 44-45, 1991.

[25] J.C. Browne. Software engineering of parallel programs in a computationally oriented display environment. In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 75–94. MIT Press, Cambridge Mass., 1990.

[26] K. M. Chandy and C. Kesselman. C++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01, California Institute of Technology, Pasadena, California.

[27] I. Foster and K. M. Chandy. Fortran M: a language for modular parallel programming. Preprint MCS-P327-0992, Argonne National Lab, 1992.

[28] F. E. Allen, M. Burke, R. Cytron, J. Ferrante, V. Sarkar and W. Hseih. A framework for determining useful parallelism. *Proceedings of the Second International Conference on Supercomputing,* St. Malo, France, July 1988.

[29] F. E. Allen, M. Burke, P. Charles and R. Cytron. An overview of the PTRAN analysis system for multiprocessing. *Parallel and Distributed Computing* 5: 617–640.

[30] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel and Distributed Systems* 3(2): 166–178, March 1992.

[31] C. D. Polychronopoulos. Toward auto-scheduling compilers. *Journal of Supercomputing* 2: 297–330, 1988.

[32] C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 252–264,Cologne, Germany, June 1991.

[33] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, Bruce P. Leung and D. A. Schouten. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 114–125. MIT Press, Cambridge Mass., 1990.

[34] N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 114–125. MIT Press, Cambridge Mass., 1990.

[35] K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. C. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise, and D. Clemmer. An environment for parallel structuring for Fortran programs. *Proc. 1989 International Conference on Parallel Processing, Volume II: Software*, pages 98–106, Chicago, Illinois, August 1989.

[36] J. K. Lee and D. Gannon. Object oriented parallel programming: experiments and results. *Proceedings of Supercomputing 91*, pages 273–282, Albuquerque, New Mexico, November 1991.