High Performance Fortran: A Practical Analysis

ALLAN KNIES[1], MATTHEW O'KEEFE[2], AND TOM MACDONALD[3]

[1]*School of Electrical Engineering, Purdue University, West Lafayette, IN 47907-1285*
[2]*Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455*
[3]*MPP Compilers, Cray Research Incorporated, Eagan, MN 55121*

ABSTRACT

The recently released high performance Fortran forum (HPFF) proposal has stirred much interest in the high performance computing industry. HPFF's most important design goal is to create a language that has source code portability and that achieves high performance on single instruction multiple data (SIMD), distributed-memory multiple instruction multiple data (MIMD), and shared-memory MIMD architectures. The HPFF proposal brings to the forefront many questions about design of portable and efficient languages for parallel machines. In this article, we discuss issues that need to be addressed before an efficient production quality compiler will be available for any such language. We examine some specific issues that are related to HPF's model of computation and analyze several implementation issues. We also provide some results from another data parallel compiler to help gain insight on some of the implementation issues that are relevant to HPF. Finally, we provide a summary of options currently available for application developers in industry.

1 INTRODUCTION

In recent years, the high performance computing market has been expanding to include many diverse and new architectures. This has made language design and implementation critical to solving software portability problems for large production application codes. To help meet this challenge, the High Performance Fortran Forum (HPFF) research group was formed and within the last year has produced a Fortran specification intended to span high performance computing. Its main goal is to retain source code portability while providing high performance across architectures as diverse as distributed memory single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD), and shared memory MIMD [1].

HPF has garnered much interest from industry due to the expectation of portability and high performance. It has also brought discussion of portable high performance languages to the forefront of parallel systems research. The strengths of the HPFF proposal are clear, broad, and high level: a simplified programming model for parallel machines, the development of a single language that can be used on many machines, a large standardized library of support routines, and a proposal developed jointly by industry and academic experts. However, it does have a number of unresolved issues that lie in the low-level details of portability, compiler development and technology,

Received September 1993
Revised January 1994

Scientific Programming, Vol. 3, pp. 187–199 (1994)
CCC 1058-9244/94/030187-13

and application implementation that are not immediately obvious. In this article, we will analyze some of these unresolved issues in the HPFF proposal as a guide for application developers and for future language development efforts.

In Section 2, we provide an informal discussion of portability and high performance to set the stage for our analysis. In Section 3, we outline general architectural issues that affect portable languages as they relate to the performance of parallel programs, In Section 4, we discuss specific problem areas in HPF as they related to the discussion in Section 3 and their affect on compiler implementation and efficiency of generated code. Section 5 discussed HPF and application implementation issue by looking at the results of the CM Fortran compiler on several application codes. Section 6 provides a brief overview of implementation possibilities for application developers now.

2 PORTABILITY AND HIGH EFFICIENCY

In section 2.2, the HPF specification notes:

> Although SIMD processor arrays, MIMD shared-memory machines, and MIMD distributed-memory machines use very different low-level primitives, there is a broad similarity with respect to the fundamental factors that affect the performance of parallel programs on these machines. Thus, achieving high efficiency across different parallel machines with the same high level HPF program is a feasible goal.

It is important to define what can be meant by high efficiency (high performance) and portability. It is possible for a language to exhibit both portability (all programs run without change on other machines) and high performance (programs written in the language can be made to run efficiently on any given machine) without exhibiting both characteristics simultaneously. We believe this distinction is important to the high performance community. Therefore, we propose an informal definition to describe what it means for a language to simultaneously exhibit both high performance and portability:

> For any system, a program written in a portable high performance language that solves a particular problem should achieve wall-clock runtime comparable to a well-written program that solves the same problem in the native high-level language on that system.

This definition says that a portable high performance language* should be expected to achieve performance similar to that already achievable on that machine by native high-level languages. Notice that if we are willing to accept portability without claiming high performance, then this definition is too stringent. The rest of this article discusses issues in the development of languages that facilitate writing codes that are simultaneously portable and efficient with emphasis on HPF.

3 ARCHITECTURAL ISSUES

In this section, we outline architectural attributes (latency, bandwidth, network topology, etc.) that are related to parallel machines. This discussion is intended to focus attention on architectural features over which a portably efficient language might allow direct control. We are not advocating that such control be allowed in a portably efficient language. Rather, we are pointing out that either the programmer or the compiler must make decisions regarding these issues and it is not clear that compilers with such analytic capabilities exist in production systems.

3.1 Bandwidth and Latency

A key feature of distributed memory architectures is the difference between remote memory bandwidth and local memory bandwidth [2]. This issue is particularly important when considering the effects of communicating large data sets between processors (i.e., when the network needs to transmit large amounts of data). Programmers trade off the cost of data redistribution versus the penalty for using a suboptimal algorithm that may be better suited for the current distribution of data. If a particular architecture can redistribute data much faster than another, then it becomes feasible for data to be redistributed rather than using a less efficient code sequence amendable to the current distribution.

* We will also use the phrase *portably efficient* to refer to portable high performance languages.

Although bandwidth affects the feasibility of large data transfers, the latency of a remote reference determines the acceptable granularity of communication. On some parallel machines, the difference between local and remote memory latency can be two orders of magnitude or more [2, 3]. In such cases, the programmer must try to minimize startup overhead by grouping data transfers into single messages. On a machine like the Cray YMP-C90, memory latency is more or less uniform for all processors and the penalty for references can sometimes be hidden by pipelined functional units (memory and computation). Such machines are capable of handling much finer grained transactions between memory and the processors. This tradeoff can be seen by studying the performance of the NAS IS benchmark [4] where the shared-memory machines do significantly better than the distributed memory machines because of their ability to do low-latency fine-grained memory accesses.

Given this tradeoff, how should the granularity of communications be factored in for a given machine: If the latency is very low, then a fine-grained approach might yield the best results, but if the latency is very high, then coarse-grained messages might be better.

3.2 Node Level Issues

The effect of memory latency is also an issue at the node level as the performance of all current microprocessors is heavily dependent on keeping data in cache. In some cases, cache performance may be the limiting performance factor for microprocessor-based machines. A good programmer takes into account blocking factors based on the size and configuration of the cache when writing loops. This has some affect on the size of messages used to communicate between processors while executing such loops. This area is particularly critical on microprocessors where multiple cache misses [5] can cause the processor to stall until one or more of the outstanding memory reads is resolved. The degree to which local data accesses can be optimized greatly affects the performance of an individual processor and hence the system as a whole.

This issue becomes more difficult as the size of the problem increases and less of the data will fit into cache. In such cases, completely new blocking sizes and techniques such as software pipelining are needed to keep the processor from stalling due to outstanding memory references.

Cache coherency is another important node level issue. If a distributed memory machine allows processors to directly access another processor's memory (i.e., global address space) without hardware-assisted coherence, then sophisticated analysis will be required to guarantee that a particular piece of data can be safely cached. One way to do this is to differentiate between shared and private data: Private data is not directly accessible by other processing elements (PEs) and is thus cachable; shared data is not cachable unless the compiler can determine it is safe to do so [6].

Finally, many machines have different levels of parallelism and locality (such as the Thinking Machines CM-5 [2]). Locality effects are present at multiple levels in the CM-5: within vector units, each having direct access to one of four memory banks; across groups of four vector units, controlled by a single SPARC processor; and across groups of four nodes, where the bandwidth is higher than communications with other nodes. To get good performance from such machines, the compiler must be able to determine which kind of parallelism is appropriate for a given code sequence. If a program is written in a language that does not provide support for expressing multilevel parallelism, then the compiler must make these decisions without the programmer's help.

3.3 Synchronization

Another area where system performance varies widely amongst architectures is the cost of synchronizing processors. The cost can vary from zero cycles on SIMD architectures such as the Maspar MP-2 to thousands of cycles on MIMD architectures such as the nCUBE-2 [3].

Because of this, it is important to be careful with the use of barriers and other synchronizations. If such synchronizations occur too frequently, then the performance of such codes will be good on some machines and poor on others.

3.4 Network Topology

An area that has historically received attention is the topological aspects of the interconnection network. Network conflict penalties impact the number of communication patterns that can be efficiently executed on a given machine. All parallel machines suffer from contention for blocks of memory (memory banks in the shared memory machine and access to a PE's network interface in a distributed memory machine), but the penalty

for unstructured communication varies dramatically. On a YMP-C90 there is relatively little contention in the network except for the time a bank is busy servicing a request (i.e., the delay is generally at the node, not within the network). This means that a YMP-C90 is very effective at doing gather/scatter operations provided the number of bank conflicts is not high. On the other hand, YMP-C90 memory performance will degrade much more quickly if the programmer sequentially accesses elements whose addresses are separated by a large power of two [7]. A distributed memory machine is somewhat different in that it will suffer not only from accesses to a given PE's network interface, but also intraversing the network. This means that a programmer will want accesses to be made so that network conflicts are minimized. In general, this means that communications patterns should be highly structured and not clustered in bursts that will overwhelm the network.

As an example of how this difference could cause problems, on a YMP-C90, a programmer will write code to avoid the problems associated with power of two strides to yield good memory performance. If the same program is to be optimized for a hypercube connected machine, communications are frequently made between elements that are separated by a power of two elements! Such coding practices inherently show up in the loop bounds and index expressions of array references.

3.5 Algorithm Issues

Thus far, our discussion has centered on how a particular algorithm might need to be implemented differently, independently of the algorithm selected. There are cases where the architectural parameters are so extreme that one algorithm simply is not feasible on that machine. This issue is clearly beyond the scope of a pure discussion of language, but is worth pointing out as a factor that influences any language's claim to be portable and efficient.

4 IMPLEMENTATION AND MODEL ISSUES IN HPF

4.1 HPF Overview

The HPF language is Fortran 90 with added directives, new syntax and semantics, and a set of new library routines. The directives are special comments that give the compiler additional information about processor arrangement, data distribution, data alignment, and statement independence. The new syntax includes the `FORALL` statement, and the keywords `EXTRINSIC`, `PURE`, and `NEW`. New semantics restrict the support of Fortran storage association and sequence association (e.g., `EQUIVALENCE`). New library routines include system inquiry, array, and bit manipulation intrinsic functions. HPF also defines a subset that vendors can choose to implement.

In the remainder of this section, we provide a detailed analysis of the impact of several of the features of HPF's computational model. The discussion is structured around the issues described in Section 3.

4.2 Node Level Issues

Address Computation

Every high-level programming language has implicit overhead associated with it that is not incurred by an assembly language programmer. A tradeoff is made by the programmer because the high-level language increases productivity and portability. When a feature is added to a language, the implicit overhead introduced by the feature needs to be understood. For example, there is an expectation that there is more implicit overhead associated with making a function call than inlining the executable statements inside the function. Again, there is a tradeoff. This time it is maintainability, good software engineering practices, and productivity. However, language features that have the appearance of simplicity but introduce significant implicit overhead need to be examined and understood by programmers hoping to get good performance.

This is particularly true in HPF with respect to addressing distributed objects. HPF allows arrays to be distributed across multiple processors to provide a global name space for data and a mechanism for increasing the locality of references. The set of distributions provided by HPF is very powerful: Arrays can be aligned with other arrays or with templates; alignments can specify dummy variables to indicate offsets along a dimension, dimensional transposes, collapses, or replications. The templates are then distributed onto processors. The dimensional distributions can be

blocked, cyclic, or "on-processor" (i.e., an entire dimension is distributed as a whole object).

One of the main advantages of the global name space is that it provides an intuitive model for accessing data and is relatively easy for a new programmer to understand. The alternative is a more cumbersome communication mechanism through an explicit functional interface.

The data distribution features in HPF let array references provide implicit communication when necessary, and increased locality when desired via data alignment and distribution. Because a data reference no longer provides explicit information about its location, the compiler must determine this before data can be accessed. In HPF, this adds additional implicit overhead when the compiler generates code to reference elements of a distributed array. This means HPF has additional implicit overhead on distributed memory machines that is not present on shared memory machines when referencing expressions such as:

```
A(I, J)
```

The additional overhead exists because the processor number and local address for this array element are extracted from the subscript expression. A shared memory machine need only compute an address. Furthermore, each index of the subscript (I and J) contributes a portion to both the processor number and the local address. That is, a sequence of execution time and compile time computations extract the relevant information from the indices, scale them appropriately, and produce two distinct values. The more complicated the distribution, the more implicit overhead is associated with extracting the processor number and local address. This overhead is independent of the network costs associated with actually accessing the remote reference.

Insight into this implicit overhead is gained by examining the calculations used to extract a processor number and offset for a distributed array. An example of a one-dimensional distribution is the following:

```
      REAL A(10000)
!HPF$ DISTRIBUTE A(CYCLIC(3))
```

In general, an array declaration looks like:

```
 DIMENSION A(L1:U1, L2: U2, ..., Lr: Ur)
!HPF$ DISTRIBUTE A(α1, α2, ..., αr)
```

where α_i is the distribution pattern for dimension i and is one of

```
BLOCK†
CYCLIC
CYCLIC(M)
*            i.e., nondistributed dimension.
```

Each α_i represents the distribution of a block of elements across the processors. The block size is computed as:

$$B_i = \begin{cases} \left\lceil \dfrac{(1 + U_i - L_i)}{N_i} \right\rceil & \text{if } \alpha_i = BLOCK \\ 1 & \text{if } \alpha_i = CYCLIC \\ M & \text{if } \alpha_i = CYCLIC(M) \\ 1 + U_i - L_i & \text{if } \alpha_i = * \end{cases}$$

where N_i is the number of processors over which the array elements in dimension i are distributed.

With the information from B_i and N_i about the data distribution, and given an index vector such that each I_i represents an index used in the array subscript expression, the local offset and processor number can be computed as follows:

$$\begin{aligned} \text{Offset} = W_i = \sum_{i=1}^{r} & ((((I_i - L_i) \bmod B_i) \\ & + \lfloor (I_i - L_i)/(B_i \times N_i) \rfloor \times B_i) \\ & \times \prod_{k=1}^{i-1} \lceil (1 + U_k - L_k)/N_k \rceil) \end{aligned}$$

$$\begin{aligned} \text{Processor} = P_i &= \sum_{i=1}^{r} ((\lfloor (I_i - L_i)/B_i \rfloor \bmod N_i) \times \prod_{k=1}^{i-1} N_k) \end{aligned}$$

The offset is the number of elements into the local portion of the array on a processor. The computations are shown in Fortran below. The IEXTENTS array assumes the $1 + U_k - L_k$ calculation has already been performed. The INDEXES arrays assumes the $I_i - L_i$ calculation has already been performed.

```
IPROC = 0
LA = 0
LA_FACTOR = 1
IP_FACTOR = 1
```

† There is also a BLOCK(M) distribution that is similar to the CYCLIC(M) distribution.

```
C
   DO I = 1, IRANK              ! for every index in subscript
     IX = INDEXES(I)            ! current index in subscript
     IBK = IBK_SIZES(I)         ! block size of this dim
     IP = N(I)                  ! number of procs in dim
     IBLOCK = MOD(IX, IBK)      ! extract block info
     LA = LA + IBLOCK * LA_FACTOR    !factor block info into local addr
     IX = IX / IBK                   ! right justify proc info
     IPROC = IPROC + MOD(IX, IP) * IP_FACTOR    !extract proc info
     IX =  IX / IP                   ! right justify cycle
     IP_FACTOR = IP _FACTOR * IP
     LA = LA + IX * IBK * LA_FACTOR  ! factor cycle info into local addr
     LA_FACTOR = LA_FACTOR * ((IEXTENTS(I) + IP - 1) / IP)
  ENDDO
```

The offset into the array is the value of LA and the processor number is the value of IPROC after the loop finishes execution. The above calculations do not show any additional work required if the distributed array is aligned (with the ALIGN directive) such that corresponding elements are not necessarily on the same processor. There are many optimizations that can be performed if the implementation knows more information about the distribution at compile time. There are also table lookup techniques that can also help reduce the number of calculations, but a time versus space tradeoff is made [8]. The above formulas and Fortran code show the general solution. In particular, the use of the division and remainder operations are necessary because the extents and block sizes can be arbitrary integers. On some microprocessors, these instructions are extremely expensive [5].

Such complex computations are not always necessary. As an example of a simple distribution where the array bounds are known statically and are powers of two and the dimensional distributions are simple, we will examine the following example declaration:

```
      REAL A(256, 128)
!HPF$ DISTRIBUTE A( CYCLIC(4), CYCLIC(4) )
C
C Assume: power of two array extents and simple distributions with
C         power of two block sizes and power of two PEs allocated across
C         each dimension (say 8 and 4 for a total of 32 PEs), and that all
C         quantities except the values of I and J are known
C         at compile time, and that constant folding is performed.
C
```

Now, to access the following element:

```
A(I,J)
```

the compiler generates code that computes the same result as the following code sequence:

```
IX = I
LA = MOD(I,4)                ! mask off 2 bits
IX = IX / 4                  ! shift right 2 bits
IPROC = MOD(IX,8)            ! mask off 3 bits
IX = IX / 8                  ! shift right 2 bits
LA = LA + IX * 4             ! shift left 2 bits and add
JX = J
LA = LA + MOD(JX,4) * 32     ! mask off 2 bits, shift left 5 bits and add
```

```
JX = JX / 4                    ! shift right 2 bits
IPROC = MOD(JX, 4) * 8         ! mask off 2 bits and shift left 3 bits
JX = JX / 4                    ! shift right 2 bits
LA = LA + JX * 128             ! shift left 7 bits and add
```

Note that the example above permits the use of shift and mask instructions because the sizes are all powers of two and that several constants have been folded. The normal cost of addressing distributed arrays in HPF is somewhere in between these two examples. It is important to remember that features such as relative alignment and unknown argument distributions will increase the complexity of addressing from what is shown above.

At some point the cost associated with extracting the processor and local address computation exceeds the network cost associated with accessing the remote data. This crossover point is highly machine dependent and critical to achieving high performance. For example, if the individual processor speed is several orders of magnitude faster than the network speed (this well might be the case if the network uses an Ethernet link, for example), then the overhead associated with computing the processor number and local address is still overwhelmed by the long latency overhead of the network communication. In this case a complex data distribution may pay off because increasing the locality of references is far more important than decreasing the cost of the address computation. However, another architecture may strive to decrease the difference between the processor speed and the network speed. Now the complex data distributions hinder performance because it is actually faster to perform the remote reference than to compute the processor number and local address.

Compiler analysis of loops often allows certain array references to be simplified. For example, a stride one array reference through a loop can be simplified to a pointer increment. With distributed arrays, there is a considerable performance benefit if the compiler is able to determine that array references are local to the processor. For example, if all the references are to corresponding elements of identically distributed arrays, then no communication is necessary and each processor can be assigned iterations involving strictly local references. The overhead associated with computing the processor number and local offset of these distributed arrays (e.g., as in the examples above) is greatly reduced, because the same transformations can be used that allow simple pointer increments.

However, if some array has different dimension extents, distribution, or a different reference pattern, then such transformations may not be possible. The problem associated with recognizing when references to different data elements are all local is much more difficult because elements are distributed in contiguous blocks. Different extents, distributions, and reference patterns can all cause the boundary point at which the next remote element is encountered to be different array references. Therefore, the set of local references for one array can be very different than for another array for a given loop. Fairly simple loops can be quite difficult to analyze and transform if different data distributions, extents, or reference patterns are involved. The worst case scenarios involve vector valued array references such as:

```
X(IX(I))
```

because the reference pattern is random as far as the compiler is concerned.

Private Data and Serialized Access

Another important node level issue is that of optimizing computations that reference processor local data.

Independent loops (not part of the subset) also permit certain variables to be declared as NEW variables, and is one form of local private data available in HPF. The primary limitation imposed on NEW variables is that their scope is limited to a single iteration as they effectively become undefined at the end of each iteration.

Another form of local data is available through PURE routines. A PURE routine must be side-effect free (i.e., no modification of global variables, no I/O, and no STOP statement). Because local variables cannot be distributed, specified on a SAVE statement, or data initialized, they can be stored as local private data on each processor. A PURE routine offers good access to local data when side-effect free computation is possible.

Consider a ray tracer application. Each ray can follow a path that is completely independent of the

other rays and would be naturally implemented using PURE routines. One ray tracing technique generates more rays when a particular ray hits certain objects (e.g., diffuse reflection), and adds them to a queue of rays waiting to be processed. Since PURE routines are not allowed to update global data, adding rays to a global queue becomes less straightforward and might introduce a load that is not well balanced. It may also be difficult to report error conditions inside PURE routines because I/O is not allowed.

Another node level issue that HPF compilers will need to solve is determining which portions of distributed arrays are cachable. If the machine has hardware-assisted cache coherence mechanisms, then the problem of correctness will not be as great, but poorly cached data will still suffer a large performance impact. On systems where there is little hardware support for cache coherence, the burden falls on compiler analysis. In cases where the compiler is unable to determine if caching is safe, some distributed data will be uncachable and operations that manipulate it will be limited by the speed of main memory rather than cache.

Finally, there are no mutual exclusion primitives available within HPF. Therefore, even though iterations can execute in parallel, it is not possible to serialize access to the globally addressable data. This makes it very difficult to exploit control parallelism.

4.3 Synchronization

The HPF specification uses array syntax and specified loops to indicate parallel operations. The FORALL statement is one of the primary mechanisms used to specify a parallel loop. The abstract execution model for this kind of parallelism requires every processor to participate in the execution of every statement. More precisely, no processor is allowed to continue on to the next executable statement until all results required by that statement are available. In the abstract model, synchronization points are required in between two adjacent array syntax statements and after evaluating the right-hand side of an assignment statement. On systems where synchronization is expensive, a programmer might be inclined to use algorithms that do not require as many global synchronizations. Optimizations can eliminate some unnecessary synchronization points, but HPF does not provide support for programmers to assist in the elimination of unwanted synchronization that the optimizer fails to find. The advantage of HPF's model is that it is conceptually more simple and programs are less likely to have race conditions.

One way of eliminating synchronization is through control parallelism. Control parallelism allows different processors to execute independent execution streams. Synchronization occurs explicitly through libraries and language features, or implicitly at the end of control blocks. Although PURE routines in HPF do permit a limited form of control parallelism, the primary mechanism for exploiting control parallelism is through the INDEPENDENT directive, which specifies that iterations of a loop execute independently of each other. However, the semantics of this directive are not completely specified. For example, it is unclear exactly what restrictions are placed on subroutines called within independent loops. Can independent iterations allocate globally addressable memory or do all processors have to participate in the allocation?

Finally, there are no explicit synchronization primitives in HPF and this combined with the lack of serialization primitives further limits the capacity for control parallelism.

5 HPF AND MPP APPLICATIONS

In this section we focus on application issues, especially as they relate to questions raised in Sections 3 and 4. Some experience has been gained with HPF-like languages on massively parallel machines; we discuss some of these early results and how they relate to HPF.

5.1 General Issues Relating to Production Environments

Languages Currently Used for Applications Development

Supercomputers are mainly used by scientists and engineers and the trend is likely to continue into the foreseeable future. For these systems, programs are generally written in Fortran 77 or C and scientists have been reluctant to use new languages even when they offer attractive features. This trend has shown that scientists would rather use a familiar language and one in which they can get predictable and reliable results and performance. As such, HPF definitely provides a familiar feel for those users familiar with Fortran 90.

However, before scientists are likely to rewrite their codes, they will want assurance that good compilers will be available on a wide variety of platforms and that their programs will achieve reasonable performance. Because HPF and Fortran 90 compilers will likely be less common and less efficient than Fortran 77 on workstations for some time, users may initially be required to maintain multiple versions of their codes—one in HPF and another in Fortran 77 for workstations and vector machines.

Porting Codes

Programmers are concerned both with the time necessary to port and optimize a code for a particular machine and the eventual wall-clock time necessary for a calculation to complete. To port a Fortran 77 code from a workstation or vector supercomputer to HPF will require that data layout directives be added and code reworked to fit into the data parallel model. In current MPP environments it is possible for ports of larger applications to any new language to take months, especially when the performance of the resulting ported code is critical (if lower performance is acceptable then the time to convert the application may be much less).

The interaction between the architectural and language issues described in Sections 3 and 4 often limits the coding styles that yield good performance [9, 10]; these styles must be adhered to on a particular machine to achieve reasonable performance. These styles often change between machines and even between different versions of the compiler, as has been observed on the CM-200 and CM-5 [11] and thus, portability becomes difficult. The sensitivity of performance to program style is a direct result of the need to efficiently manage many architectural features at once.

Model of Computation

Many scientific problems fit well into HPF's data parallel model of computation. For these applications, the data parallel model is intuitively appealing because arrays are first class objects. However, in some cases, scientists may find the data parallel model does not match their application requirements even though that is HPF's primary means of efficient parallel computation. For example, irregular computations, such as those required on sparse matrices, are not well supported in HPF (although the HPFF plans to consider this issue in the future).

Another limitation of the data parallel model is that it is difficult for the programmer to directly express parallelism implicit across multiple loops and subprograms. Because parallelism is expressed and optimizations are performed on a statement-by-statement basis, sophisticated compiler technology will be needed to expand the scope of this optimization. The current HPF specification provides only indirect mechanisms (local routines, `INDEPENDENT`, `EXTRINSIC`) for the programmer to indicate this global parallelism. A possible solution to this problem would be to allow more features to support control parallelism.

5.2 Applications Codes and CM Fortran

CM Fortran Overview

CM Fortran [12] represents an HPF-like language that has been implemented on both SIMD (CM-200) and MIMD (CM-5) machines. Experience has been gained on the performance of several interesting application codes written in CM Fortran [4, 13]; this experience can indicate potential performance of HPF implementations.

CM Fortran includes the array syntax of Fortran 90, a variant of the `FORALL` construct, and data layout and alignment directives. These directives are similar but not as general as those found in HPF; in particular, an array dimension is either "on-processor" (contained in a single processor) or "parallel" (spread across all processors). In HPF, the `BLOCK` specification allows contiguous portions of an array axis to be spread across multiple processors. With this notable exception, CM Fortran's model is quite close to subset HPF [1].

Applications Characteristics

The CM Fortran codes we discuss are based on finite-difference and finite-volume methods [14], which are commonly employed to solve a variety of partial differential equations representing physical processes, including high Mach number fluid flows [15], mesoscale whether phenomena [16], and ocean circulation [17]. These numerical methods yield application codes that are particularly well suited for distributed memory architectures.

These applications codes are often characterized by logically regular grids representing various state variables such as pressure, density, temperature, and velocities; these programs step (integrate) forward in time, updating the state variables using appropriate equations of state. These regu-

lar grids can be readily decomposed into independent patches or bricks that can be mapped directly to processors; operations within each patch or brick proceed independently on local data for most of the time integration step. This grid model is common to many applications in high performance computing and exhibits large amounts of parallelism and locality both across and within nodes.

Piecewise Parabolic Method

An example of this kind of code is the Piecewise Parabolic Method (PPM), a hydrodynamics code used to study high Mach number compressible flows with strong shocks and other nonlinear interactions. Several of the current versions of PPM [15] have been translated to CM Fortran, attaining about 6.5 gigaflops (Gflops) on large 2-D grids (16 million zones) on the 512-node, 16-gigabyte (GByte) main memory CM-5 at the University of Minnesota. These codes are approximately 5,000 lines of clean, almost completely vectorizable Fortran 77. The CM-5 versions of PPM employs a domain decomposition where the grid is partitioned into subdomains that are mapped directly to processors. PPM has been written and translated so that communication is required only once per time step; all communication is isolated in one small subroutine. At the end of each time step, extra "fake" zones are updated both on the grid boundaries (to implement boundary conditions) and on interior patches (to reduce the need for communication during the time step by maintaining redundant copies).

Although limited, these communication operations tend to dominate PPM execution time on the CM-5 and limit speedup even for large grids. Various strategies have been employed to reduce the overhead found in the CM Fortran run-time system for these operations, including copying array boundary sections in static arrays (using a special routine that suppresses off-chip communication), transferring these between processors, then copying the static array's values back into the state variable arrays. This approach reduces the current run-time system overhead necessary for the most straightforward transfer and reduces communication execution time by a factor of four.

The ARPS

The ARPS is a meso-scale prediction Fortran 77 code under development by the Center for Analysis and Prediction of Storms (CAPS) at the University of Oklahoma. ARPS is a 3-D, fully compressible, nonhydrostatic weather model that includes cloud physics (humidity, cloud water, and rain water), surface effects, and subgrid-scale turbulent mixing.‡ The code for the model is over 50,000 lines of modular Fortran 77 [16].

This code has recently been translated to CM Fortran by a group at Thinking Machines (using CMAX, a Fortran 77 to CM Fortran translator [18]) and by the Fortran P group at Minnesota [19]. The Thinking Machines group was able to achieve approximately 3 Gflops on a 512-node CM-5 on this code, which is impressive in the context of automatic translation but is much less than the peak speed of the machine. Various run-time overheads could be playing a significant role in this performance result. The Fortran P group encountered memory inefficiencies related to earlier versions of the CM Fortran compiler that made it difficult to achieve efficient execution on the translated code. More recent releases of the compiler seem to have removed some of these problems [19].

5.3 Potential Overheads in HPF Applications

In the previous section, we described the characteristics of several applications that have been implemented in CM Fortran. While executing these translated codes, several overheads associated with CM Fortran have been observed.

For example, the local nature, in a lexical sense, of data parallel computations can result in the generation of large intermediate temporary arrays during expression evaluation. Temporary arrays are often generated at subroutine boundaries when array sections are used as actual arguments; the CM Fortran compiler would often generate communication in these cases. Solutions such as subroutine cloning are possible for the latter case.

Additionally, communication and run-time overheads found in CM Fortran have been seen to play a role in performance problems in these codes. In particular, the compiler is sometimes unable to determine whether a memory access is local or off-processor, despite explicit data layout directives. When uncertain, the compiler must generate more general off-processor communica-

‡ The ARPS uses second-order quadratically conservative spatial discretization and second-order leapfrog (with Asselin time filter option) temporal discretization, based on an Arakawa C-grid and terrain-following vertical coordinates.

tion operations even if the data exists locally. These operations are implemented in a run-time system library called during program execution. In any case, the run-time system must first decide if the communication is local or note. On the CM-200, this phenomena often greatly reduced performance for certain CM Fortran coding styles [9].

In summary, these studies suggest that HPF-like languages can be used to express typical numerical codes. However, due to limited expressiveness of CM Fortran and significant implicit overheads associated with the language and its compiler, applications have yielded significantly less than peak performance on machines such as the CM-5. Of course, this does not preclude the possibility of constructing an HPF compiler and run-time system that can achieve high efficiency. However, given the amount of work required in porting CM Fortran to the CM-5 by a well-established and experienced MPP company such as Thinking Machines, it is clear that engineering an efficient HPF compiler will be challenging.

6 CURRENT PROGRAMMING SYSTEMS AND THE FUTURE

Presently, we believe that there are no portably efficient languages available. Parallel machine architectures are changing rapidly relative to single CPU architectures, so any language design effort based on specific assumptions about current machine architecture may be obsolete before it can become widespread. As such, application developers need to prioritize their time and effort.

For those who seek portability, message passing is well established and currently provides the best performance for many machines. Although message passing leaves many problems unsolved (such as when and how to redistribute data) and is considered by many to be more difficult to program than shared memory models, it does provide a well-understood path towards portability and reasonable performance. Additionally, because it does not support a global namespace, it does not suffer from the side effects of a global name space (cache coherency, hidden data access latency variations, addressing complexity). Message passing also cleanly exploits all the resources expended on optimizing compilers for single processors.

Recently, several very large codes have been successfully ported using this approach. The Integrated Forecast System (IFS) medium-range weather prediction model developed by the ECMWF [20] consists of nearly 220,000 lines of Fortran 77; only a very small port of the code required changes to add message-passing calls.§ The ECMWF and Cray Research have recently ported the full 3-D IFS model to the Cray T3D massively parallel processor.

Similarly, message passing versions of PPM have been developed by Woodward and his colleagues that isolate the message passing code to one small routine. The most recent message passing version of PPM has been ported to a cluster of 16 Challenge XL servers‖ from Silicon Graphics and achieved nearly 5 Gflops [21]. These efforts show that with discipline in organizing and designing code message passing can be an easy-to-use and effective programming paradigm.

However, message passing is not the final answer to portable and efficient parallel programming. Newer machines such as the Cray T3D, KSR-1, and KSR-2, and announced products from Convex and Tera Computer all have some form of hardware support for fetching remote data via a common address space. In this context, the message-passing model adds a level of abstraction between the programmer and the hardware, which adds significant unnecessary software overhead on remote data accesses.

Another near-term solution is the use of standard, vendor supplied, highly optimized, libraries of commonly used routines. This is very effective for problems with simple structure that can be mapped to the machine using simple rules. However, if a problem requires that data be distributed in a complex or unusual way, it is unlikely that the library will support all the necessary data mappings. Another drawback to simple library-based schemes is that it is very difficult to express nested parallelism given a fixed functional interface. The most obvious problem with library routines is that they are only useful if the needed routines are in the library.

In the future, automatic translation of serial languages for massively parallel machines is the most desirable solution. We expect that automatic translation will continue to improve via new compilation techniques [22, 23], but it is optimistic to assume they will be able to get performance comparable to explicitly parallel languages on MPPs.

§ Message-passing calls were required when transposing between the different grid spaces employed in the model.

‖ Each Challenge XL server in the configuration had 20 100-MHz R4400 CPUs, 1.75 GByte of memory, 12 GByte of local disk, 1 Exabyte tape drive, and 3 FDDI interfaces to a 3-D toroidal network.

However, if it is possible for programmers to limit themselves to a self similar coding style, the Fortran P translator developed at Minnesota can translate Fortran 77 codes to MPP form, allowing the user to maintain and modify the original Fortran 77 yet exploit new hardware platforms [11]. Similarly, the CMAX translator developed by Thinking Machines [18] also translates a subset of Fortran 77 (known as Scalable Fortran) and has been applied successfully to several codes.

HPF also has a number of features that make it attractive in the near term. HPF is standardizing the data parallel approach in the same manner that MPI [24] is standardizing message passing. Also, the simplicity of the data parallel model and a global address space makes programming in HPF relatively easy as it relieves the programmer from explicitly specifying communications and remote addressing. HPF also provides a single source language that will be able to run on many different architectures even if individual programs may not port efficiently. It also tends to localize the source code changes necessary when porting to the data layout directives. Finally, HPF has proposed a large library of routine interfaces to be part of the language, thus guaranteeing programmers availability of these routines on systems with HPF compilers.

7 CONCLUSIONS

HPF is a well-conceived, well-documented language proposal that will help future research explore the issues surrounding language design for parallel machines.

However, although the HPF specification claims that HPF programs can achieve both portability and high performance—there is some reason for doubt. Different parallel architectures have very different overheads that make it difficult to try to balance one kind of overhead versus another in a portable source level program in any language.

There are a large number of implicit synchronization points in HPF due to its data parallel model. This raises concerns about performance and the limitations placed on control parallelism and local private data. Furthermore, the data distribution mechanisms available in HPF permit very complex distributions that can introduce significant implicit overhead in the computation of array element addresses as discussed in this article. It is unknown whether the flexibility gained by such complex distributions will outweigh the overhead they introduce.

We have raised many questions about HPF in this article, and we do not claim that they cannot be overcome, but rather that there are many open issues that are unresolved. These questions can only be answered after HPF compilers are available and tested—this can be done without the high performance industry adopting HPF as a standard before it has been proven. Additionally, as HPF compilers are developed and performance results obtained, new questions will be raised. Answers to these questions will help identify the language features that should be present in a standard high performance language. The problem is further complicated by the rapidly evolving state of parallel architectures and their changing strengths and weaknesses. Thus, it is important that HPF and other new languages be implemented and carefully studied to avoid mistakes that could have been averted so that the best solutions can be adopted as standards in the future.

ACKNOWLEDGMENTS

We would like to thank Prof. Paul Woodward and Steve Anderson at the University of Minnesota for their many insights on using CM Fortran and PPM performance on the CM-5. Further thanks go to Woody Lichtenstein and Gary Sabot of Thinking Machines for their help and insights on the CM Fortran compiler and run-time system. We would also like to thank George Adams for his insights and comments on early drafts of this article. Finally, we would like to thank the referees for their comments and recommendations. Matthew O'Keefe was supported in part by the Office of Naval Research grant no. N00014-93-1-0426 and by contract no. DAAL02-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center.

REFERENCES

1. "High Performance Fortran Language Specification," High Performance Fortran Forum, *Scientific Programming*, Vol. 2, Nos. 1 & 2, 1993.
2. Thinking Machines Corporation, *Connection Machine Model CM-5, Technical Summary*, October 1991.
3. nCUBE Corporation, *nCUBE2 Programmer's Reference Manual*, 1990.

4. D. Bailey, E. Barsczc, L. Dagum, and H. Simon, *Supercomputing '92*. Los Alamitos, CA: IEEE Computer Society Press, 1992. pp. 386–393.
5. Digital Equipment Corpo., *21064-AA RISC CPU Microprocessor Release Notes*, 1992.
6. D. Pase, T. MacDonald, and A. Meltzer, *MPP Fortran Programming model*. Eagan, MN: Cray Research, Inc., 1993.
7. Cray Research Incorporated, *YMP-C90 Programmer's Reference Manual*, 1991.
8. S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S.-H. Teng, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego. CA. New York: ACM, 1993, pp. 149–158.
9. G. Sabot, "CM Fortran optimization notes: Slicewise model," Thinking Machines Corp., Technical Report TMC-184, March 1991.
10. H. G. Dietz, M.T. O'Keefe, T.J. Parr, T. Varghese, and P.R. Woodward, "Fortran-P," *University of Minnesota Supercomputer Institute Technique Report*, December 1991.
11. M. O'Keefe, T. Parr, B.K. Edgar, S. Anderson, P. Woodward, and H. Dietz, "The Fortran-P translator: Automatic translation of Fortran 77 programs for massively parallel processors," Army High Performance Computing Center Preprint no. 93-021.
12. Thinking Machines Corporation, *CM Fortran Reference Manual*, 1991.
13. O. Lubeck, M. Simmons, and H. Wasserman, *Supercomputing '92*. Los Alamitos, CA: IEEE Computer Society Press, 1992, pp. 403–412.
14. R. D. Richtmeyer and K.W. Morton, *Difference Methods for Initial-Value Problems*. New York: John Wiley and Sons, second edition, 1967.
15. P. R. Woodward, *Astrophysical Radiation Hydrodynamics*. D. Reidel Publishing Co., 1986, pp. 245–326.
16. K. Droegemeier, K. Johnson, K. Mills, and M. O'Keefe, *Proceedings of the 5th Workshop on the Use of Parallel Processors in Meteorology*. Reading, England. Singapore: World Scientific Publishing, Ltd., pp. 99–129, 1992.
17. R. Bleck and L. Smith, "A wind-driven isopycnic coordinate model of the North and Equatorial Atlantic Ocean," *J. Geophys. Res.*, vol. 95, pp. 3273–3285, 1990.
18. G. W. Sabot and S. Wholey, *Proceedings of the 7th International Conference on Supercomputing*. Tokyo, Japan. Los Alamitos, CA: IEEE Computer Society Press, 1993.
19. A. Sawdey and M. O'Keefe, *Proceedings of the Conference on HPC in Geosciences*. Les Houches, France: Kluwer, in press.
20. D. Dent, *Proceedings of the 5th ECMWF Workshop on the Use of Parallel Processors in Meteorology*. Reading, England. Singapore: World Scientific Publishing, Ltd., pp. 73–87, 1992.
21. D. Porter, P. Woodward, S. Anderson, K. Chin-Purcell, R. Hessel, D. Perro, I. Zacharov, J. Ryan, L Widra, and M. Galles, "Attacking a grand challenge in computational fluid dynamics on a cluster of silicon graphics challenge machines," Technical Report, University of Minnesota, November 1993.
22. M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, pp. 179–193, 1992.
23. S. Amarasinghe and M. Lam, *Proceedings of the 1993 SIGPLAN Conference on Programming Language, Design and Implementation*. Albuquerque, NM. New York: ACM, 1993, pp. 126–138.
24. Message Passing Interface Forum, "DRAFT: Document for a Standard Message Passing Interface," August 14, 1993.