The Rise and Fall of High Performance Fortran: An Historical Object Lesson

Ken Kennedy Charles Koelbel
Rice University, Houston, TX
{ken,chk}@rice.edu

Hans Zima
Institute of Scientific Computing, University of Vienna, Austria, and
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA
zima@jpl.nasa.gov

Abstract

High Performance Fortran (HPF) is a high-level data-parallel programming system based on Fortran. The effort to standardize HPF began in 1991, at the Supercomputing Conference in Albuquerque, where a group of industry leaders asked Ken Kennedy to lead an effort to produce a common programming language for the emerging class of distributed-memory parallel computers. The proposed language would focus on data-parallel operations in a single thread of control, a strategy which was pioneered by some earlier commercial and research systems, including Thinking Machines' CM Fortran, Fortran D, and Vienna Fortran.

The standardization group, called the High Performance Fortran Forum (HPFF), took a little over a year to produce a language definition that was published in January 1993 as a Rice technical report [50] and, later that same year, as an article in Scientific Programming [49].

The HPF project had created a great deal of excitement while it was underway and the release was initially well received in the community. However, over a period of several years, enthusiasm for the language waned in the United States, although it has continued to be used in Japan.

This paper traces the origins of HPF through the programming languages on which it was based, leading up to the standardization effort. It reviews the motivation underlying technical decisions that led to the set of features incorporated into the original language and its two follow-ons: HPF 2 (extensions defined by a new series of HPFF meetings) and HPF/JA (the dialect that was used by Japanese manufacturers and runs on the Earth Simulator).

A unique feature of this paper is its discussion and analysis of the technical and sociological mistakes made by both the language designers and the user community:, mistakes that led to the premature abandonment of the very promising approach employed in HPF. It concludes with some lessons for the future and an exploration of the influence of ideas from HPF on new languages emerging from the High Productivity Computing Systems program sponsored by DARPA.

Categories and Subject Descriptors K.2 History of Computing [*Software*]

General Terms Languages and Compilers, Parallel Computing

Keywords High Performance Fortran (HPF)

1. Background

Parallelism—doing multiple tasks at the same time—is fundamental in computer design. Even very early computer systems employed parallelism, overlapping input-output with computing and fetching the next instruction while still executing the current one. Some computers, like the CDC 6600, used multiple instruction execution units so that several long instructions could be in process at one time. Others used *pipelining* to overlap multiple instructions in the same execution unit, permitting them to produce one result every cycle even though any single operation would take multiple cycles. The idea of pipelining led to the first *vector computers*, exemplified by the Cray-1 [31], in which a single instruction could be used to apply the same operation to arrays of input elements with each input pair occupying a single stage of the operation pipeline. Vector machines dominated the supercomputer market from the late 1970s through the early 1980s.

By the mid-1980s, it was becoming clear that *parallel computing*, the use of multiple processors to speed up a single application, would eventually replace, or at least augment, vector computing as the way to construct leading-edge supercomputing systems. However, it was not clear what the right high-level programming model for such machines would be. In this paper we trace the history of High Performance Fortran (HPF), a representative of one of the competing models, the *data-parallel* programming model. Although HPF was important because of its open and public standardization process, it was only one of many efforts to develop parallel programming languages and models that were active in the same general time period. In writing this paper, we are *not* attempting comprehensively to treat all this work, which would be far too broad a topic; instead we include only enough material to illustrate relationships between the ideas underlying HPF and early trends in parallel programming.

We begin with a narrative discussion of parallel computer architectures and how they influenced the design of programming models. This leads to the motivating ideas behind HPF. From there, we cover the standardization process, the features of the language, and experience with early implementations. The paper concludes with a discussion of the reasons for HPF's ultimate failure and the lessons to be learned from the language and its history.

1.1 Parallel Computer Systems

Several different types of parallel computer designs were developed during the formative years of parallel processing. Although we present these architectures in a sequential narrative, the reader should bear in mind that a variety of computers of all the classes were always available or in development at any given time. Thus, these designs should be viewed as both contemporaries and competitors.

Data-Parallel Computers The *data-parallel computation model* is characterized by the property that sequences of operations or statements can be performed in parallel on each element of a collection of data. Some of the earliest parallel machines developed in the 1960s, such as the Solomon [87] and Illiac IV [9] architectures, implemented this model in hardware. A single control unit executed a sequence of instructions, broadcasting each instruction to an array of simple processing elements arranged in a regular grid. The processing elements operated in lockstep, applying the same instruction to their local data and registers. Flynn [37] classifies these machines as *Single-Instruction Multiple-Data (SIMD)* architectures. Once you have a grid of processors, each with a separate local memory, data values residing on one processor and needed by another have to be copied across the interconnection network, a process called *communication*. Interprocessor communication of this sort causes long delays, or *latency*, for cross-processor data access.

The vector computers emerging in the 1970s, such as the Cray-1, introduced an architecture paradigm supporting a simple form of data parallelism in hardware. As with the original SIMD architectures, vector computers execute a single thread of control; a key difference is the increased flexibility provided to the programmer by abandoning the need to arrange data according to a hardware-defined processor layout. Furthermore, a vector processor does not experience the problems of communication latency exhibited by more general SIMD processors because it has a single shared memory.

In the 1980s, advances in VLSI design led to another generation of SIMD architectures characterized by thousands of 1-bit processing elements, with hardware support for arbitrary communication patterns. Individual arithmetic operations on these machines were very slow because they were performed in "bit serial" fashion, that is, one bit a time. Thus any performance improvement came entirely from the high degrees of parallelism. Important representatives of this class of machines include the Connection Machines CM-2 and CM-200 [52] as well as the MasPar MP-1 [27].

The programming models for SIMD machines emphasized vector and matrix operations on large arrays. This was attractive for certain algorithms and domains (e.g., linear algebra solvers, operations on regular meshes) but confining in other contexts (e.g. complex Monte Carlo simulations). In particular, this model was seen as intuitive because it had a single thread of control, making it similar to the sequential programming model As we will see, this model strongly influenced the design of data-parallel languages.

The principal strength of data-parallel computing—fully synchronous operation—was also its greatest weakness. The only way to perform conditional computations was to selectively turn off computing by some processors in the machine, which made data-parallel computers ill suited to problems that lacked regularity, such as sparse matrix processing and calculations on irregular meshes. An additional drawback was the framework for communicating data between processors. In most cases, the processors were connected only to nearest neighbors in a two- or three-dimensional grid. [1] Because all operations were synchronous, it could take quite a long time to move data to distant processors on a grid, making these machines slow for any calculation requiring long-distance communication. These drawbacks led to the ascendance of asynchronous parallel computers with more flexible interconnection structures.

Shared-Memory Asynchronous Parallel Computers A key step in the emergence of today's machines was the development of architectures consisting of a number of full-fledged processors, each capable of independently executing a stream of instructions. At first, the parallel programming community believed that the best design for such *Multiple-Instruction Multiple-Data (MIMD)* multiprocessing systems (in the Flynn classification) should employ some form of hardware shared memory, because it would make it easy to implement a shared-memory programming model, which

[1] Important exceptions include STARAN [10] with its "flip" network, the CM-1, CM-2 and CM-200 [52] with their hypercube connections, and the MasPar MP-1 [27] with its Global Router.

was generally viewed as the most natural from the programmer's perspective. In this model, the independent processors could each access the entire memory on the machine. A large number of shared-memory multiprocessors, which are today referred to as *symmetric multiprocessors (SMPs)*, appeared as commercial products. Examples of such systems include the Alliant FX series, CDC Cyber 205, Convex C series, Cray XMP and YMP, Digital VAX 8800, Encore Multimax, ETA-10, FLEX/32, IBM 3090, Kendall Square KSR1 and KSR2, Myrias SPS-1 and SPS-2, and the Sequent Balance series [5, 30, 97, 56, 59, 62, 71, 74, 48, 32]. It should be noted that several of the machines in this list (e.g., the Convex, Cray, ETA, and IBM systems) were hybrids in the sense that each processor could perform vector operations. Many of today's multi-core architectures can be considered descendents of SMPs.

In the late 1980s, the conventional view was that shared-memory multiprocessors would be easy to program while providing a big performance advantage over vector computers. Indeed, many thought that shared memory multiprocessing would yield computing power that was limited only by how much the user was willing to pay. However, there were two problems that needed to be resolved before this vision could be realized.

The first problem was *scalability*: how such systems could be scaled to include hundreds or even thousands of processors. Because most of the early SMPs used a bus—a single multi-bit data channel that could be multiplexed on different time steps to provide communication between processors and memory—the total bandwidth to memory was limited by the aggregate number of bits that could be moved over the bus in a given time interval. For memory-intensive high-performance computing applications, the bus typically became saturated when the number of processors exceeded 16. Later systems would address this issue through the use of crossbar switches, but there was always a substantive cost, either in the expense of the interconnect or in loss of performance of the memory system.

The second problem was presented by the programming model itself. Most vendors introduced parallel constructs, such as the *parallel loop*, that, when used incorrectly, could introduce a particularly nasty form of bug called a *data race*. A data race occurs whenever two parallel tasks, such as the iterations of a parallel loop, access the same memory location, with at least one of the tasks writing to that location. In that case, different answers can result from different parallel schedules. These bugs were difficult to locate and eliminate because they were not repeatable; as a result debuggers would need to try every schedule if they were to establish the absence of a race. Vector computers did not suffer from data races because the input languages all had a single thread of control: it was up to the compiler to determine when the operations in a program could be correctly expressed as vector instructions.

Distributed-Memory Computers The scalability problems of shared memory led to a major change in direction in parallel computation. A new paradigm, called *distributed-memory parallelism* (or more elegantly, *multicomputing*), emerged from academic (the Caltech Cosmic Cube [85] and Suprenum [41]) and commercial research projects (Transputer networks [92]). In distributed-memory computers, each processor was packaged with its own memory and the processors were interconnected with networks, such as two-dimensional and three-dimensional meshes or hypercubes, that were more scalable than the bus architectures employed on shared-memory machines.

Distributed memory had two major advantages. First, the use of scalable networks made large systems much more cost-effective (at the expense of introducing additional latency for data access). Second, systems with much larger aggregate memories could be assembled. At the time, almost every microprocessor used 32-bit (or smaller) addresses. Thus a shared-memory system could only address 2^{32} different data elements. In a distributed-memory system, on the other hand, each processor could address that much memory. Therefore, distributed memory permitted the solution of problems requiring much larger memory sizes.

Unsurprisingly, the advantages of distributed memory came at a cost in programming complexity. In order for one processor to access a data element in another processor, the processor in whose local memory the data element was stored would need to *send* it to the processor requesting it; in turn, that processor would need to *receive* the data before using it. Sends and receives had to be carefully synchronized to ensure that the right data was communicated, as it would be relatively easy to match a receive with the wrong send. This approach, which was eventually standardized as *Message Passing Interface (MPI)* [70, 88, 42], requires that the programmer take complete responsibility for managing and synchronizing communication. Thus, the move to distributed memory sacrificed the convenience of shared memory while retaining the complexity of the multiple threads of control introduced by SMPs.

As a minor simplification, the standard programming model for such systems came to be the *Single-Program Multiple Data (SPMD)* model [58, 34], in which each processor executed the same program on different portions of the data space, typically that portion of the data space owned by the executing processor. This model, which is an obvious generalization of the SIMD data-parallel model, required the implementation to explicitly synchronize the processors before communication because they might be working on different parts of the program at any given time. However, it allows the effective use of control structures on a per-processor basis to handle processor-local data in between communication steps and permits the programmer to focus more narrowly on communications at the boundaries of each processor's data space. Although this is a major improvement, it does not completely eliminate the burden of managing communication by hand.

1.2 Software Support for Parallel Programming

In general, there were three software strategies for supporting parallel computing: (1) automatic parallelization of sequential languages, (2) explicitly parallel programming models and languages, and (3) data-parallel languages, which represented a mixture of strategies from the first two. In the paragraphs that follow we review some of the most important ideas in these three strategies with a particular emphasis on how they were used to support data parallelism, the most common approach to achieving scalability in scientific applications.

Automatic Parallelization The original idea for programming shared-memory machines was to adapt the work on automatic vectorization which had proven quite successful for producing vectorized programs from sequential Fortran[2] specifications [3, 99]. This work used the theory of *dependence* [65, 63, 4], which permitted reasoning about whether two distinct array accesses could reference the same memory location, to determine whether a particular loop could be converted to a sequence of vector assignments. This conversion involved distributing the loop around each of the statements and then rewriting each loop as a series of vector-register assignments. The key notion was that any statement that depended on itself, either directly or indirectly, could not be rewritten in this manner.

Vectorization was an extremely successful technology because it focused on inner loops, which were easier for programmers to understand. Even though the so-called "dusty-deck Fortran" programs did not always vectorize well, the user could usually rewrite those codes into "vectorizable loop" form, which produced good behavior across a wide variety of machines. It seemed very reasonable to assume that similar success could be achieved for shared-memory parallelization.

Unfortunately, SMP parallelism exhibited additional complexities. While vector execution was essentially synchronous, providing a synchronization on each operation, multiprocessor parallelism required explicit synchronization operations (e.g. barriers or event posting and waiting), in addition to the costs of task startup and the overhead of data sharing across multiple caches. To compensate for these costs, the compiler either needed to find very large loops that could be subdivided into large chunks, or it needed to parallelize outer loops. Dependence analysis, which worked well for vectorization, now needed to be applied over much larger loops, which often contained subprogram invocations. This led researchers to focus on interprocedural analysis as a strategy for determining whether such loops could be run in parallel [93, 16, 29].

By using increasingly complex analyses and transformations, research compilers have been able to parallelize a number of interesting applications, and commercial compilers are routinely able to parallelize programs automatically for small numbers of processors. However, for symmetric multiprocessors of sufficient scale, it is generally agreed that some form of user input is required to do a good job of parallelization.

The problems of automatic parallelization are compounded when dealing with distributed memory and message-passing systems. In addition to the issues of discovery of parallelism and granularity control, the compiler must determine how to lay out data to minimize the cost of communication. A number of research projects have attacked the problem of automatic data layout [67, 60, 6, 23, 44], but there has been little commercial application of these strategies, aside from always using a standard data layout for all arrays (usually block or block-cyclic).

Explicitly Parallel Programming: PCF and OpenMP Given the problems of automatic parallelization, a faction of the community looked for simple ways to specify parallelism explicitly in an application. For Fortran, the first examples appeared on various commercial shared-memory multiprocessors in the form of parallel loops, parallel cases, and parallel tasks. The problem with these efforts was lack of consistency across computing platforms. To overcome this, a group convened by David J. Kuck of the University of Illinois and consisting of researchers and developers from academia, industry, and government laboratories began to develop a standard set of extensions to Fortran 77 that would permit the specification of both loop and task parallelism. This effort came to be known as the *Parallel Computing Forum (PCF)*. Ken Kennedy attended all of the meetings of this group and was deeply involved in drafting the final document that defined PCF Fortran [66], a single-threaded language that permitted SPMD parallelism within certain constructs, such as parallel loops, parallel case statements, and "parallel regions." A "parallel region" created an SPMD execution environment in which a number of explicitly parallel loops could be embedded. PCF Fortran also included mechanisms for explicit synchronization and rules for storage allocation within parallel constructs.

The basic constructs in PCF Fortran were later standardized by ANSI Committee X3H5 (Parallel Extensions for Programming Languages) and eventually found their way into the informal standard for OpenMP [73, 33].

The main limitation of the PCF and OpenMP extensions is that they target platforms—shared-memory multiprocessors with uniform memory access times—that have been eclipsed at the high end by distributed-memory systems. Although it is possible to generate code for message-passing systems from OpenMP or PCF Fortran, the user has no way

[2] Although the official standards changed the capitalization from "FORTRAN" to "Fortran" beginning with Fortran 90, we use the latter form for all versions of the language.

of managing communication or data placement at the source level, which can result in serious performance problems.

Data-parallel Languages One way to address the shortcomings of the PCF/OpenMP model is explicit message passing. In this approach, a standard language (such as Fortran or C/C++) is extended with a message-passing library (such as MPI), providing the programmer with full control over the partitioning of data domains, their distribution across processors, and the required communication. However, it was soon understood that this programming paradigm can result in complex and error-prone programs due to the way in which algorithms and communication are inextricably interwoven.

This led to the question of whether the advantages of shared memory, and even a single thread of control, could be *simulated* on a distributed-memory system. A second important question was how parallelism could be made to scale to hundreds or thousands of processors. It was clear that addressing the second question would require exploiting the data-parallel programming model: by subdividing the data domain in some manner and assigning the subdomains to different processors, the degree of parallelism in a program execution is limited only by the number of processors, assuming the algorithm provides enough parallelism. With more available processors, a larger problem can be solved.

These issues led researchers to explore the new class of *data-parallel languages*, which were strongly influenced by the SIMD programming paradigm and its closeness to the dominating sequential programming model. In data-parallel languages, the large data structures in an application would be laid out across the memories of a distributed-memory parallel machine. The subcomponents of these distributed data structures could then be operated on in parallel on all the processors. Key properties of these languages include a global name space and a single thread of control through statements at the source level, with individual parallel statements being executed on all processors in a (loosely) synchronous manner.[3] Communication is not explicitly programmed, but automatically generated by the compiler/runtime system, based on a declarative specification of the data layout.

Any discussion of data-parallel languages should include a discussion of Fortran 90, because it was the first version of Fortran to include array assignment statements. A Fortran 90 array assignment was defined to behave as if the arrays used on the right-hand side were all copied into unbounded-length vector registers and operated upon before any stores occurred on the left. If one considers the elements of an infinite-length vector register as a distributed memory, then in a very real sense, a Fortran 90 array assignment is a data-parallel operation. In addition, if you had a large SIMD machine, such as the Thinking Machines CM-2, multidimensional array assignments could be also be executed in parallel. It is for this reason that Fortran 90 was so influential on the data-parallel languages to follow. However, with the advent of more complex distributed-memory systems, achieving good Fortran 90 performance became more challenging, as the data distributions needed to take both load balance and communication costs into account, along with the limitations of available storage on each processor.

[3] A computation is said to be *loosely synchronous* if it consists of alternating phases of computation and interprocessor communication.

A number of new data-parallel languages were developed for distributed-memory parallel machines in the late 1980s and early 1990s, including *Fortran D* [38, 53], *Vienna Fortran* [102, 21], *CM Fortran* [91], *C** [45], *Data-Parallel C*, *pC++* [14] and *ZPL* [89, 19]. Several other academic as well as commercial projects also contributed to the understanding necessary for the development of HPF and the required compilation technology [7, 47, 79, 68, 69, 77, 78, 80, 81, 95, 57, 51, 75].

To be sure, the data-parallel language approach was not universally embraced by either the compiler research or the application community. Each of the three strategies for software support described in this section had their passionate adherents. In retrospect, none of them has prevailed in practice: explicit message-passing using MPI remains the dominant programming system for scalable applications to this day. The main reason is the complexity of compiler-based solutions to the parallelization, data layout, and communication optimization problems inherent, to varying degrees, in each of the three strategies. In the rest of this paper, we narrow our focus to the data-parallel approach as embodied in the HPF extensions to Fortran. In the discussion that follows, we explore the reasons for the failure of HPF in particular. However, many of the impediments to the success of HPF are impediments to the other approaches as well.

2. HPF and Its Precursors

In the late 1980s and early 1990s, Fortran was still the dominant language for technical computing, which in turn was the largest market for scalable machines. Therefore, it was natural to assume that a data-parallel version of Fortran would be well received by the user community, because it could leverage the enormous base of software written in that language. Two research languages, Fortran D and Vienna Fortran, and one commercial product, CM Fortran, were among the most influential data-parallel languages based on Fortran and deeply influenced the development of HPF. In this section we review the salient issues in each of these languages. The section then concludes with a description of the activities leading to the HPF standardization effort, along with an overview of the effort itself.

Fortran D In 1987, the research group led by Ken Kennedy at Rice began collaborating with Geoffrey Fox on how to support high-level programming for distributed-memory computers. Fox had observed that the key problem in writing an application for a distributed-memory system was to choose the right data distribution, because once that was done, the actual parallelism was determined by the need to minimize communications. Thus, computations would be done in parallel on the processors that owned the data involved in that computation. This led to the idea that a language with shared memory and a single thread of control could be compiled into efficient code for a distributed-memory system if the programmer would provide information on how to distribute data across the processors. Fox and the Rice group produced a specification for a new language called Fortran D that provided a two-level distribution specification similar to that later adopted into HPF. The basic idea was that groups of arrays would be aligned with an abstract object called a *template*, then all of these arrays would be mapped to processors by a single distribution statement that mapped the template to those processors. Fortran D supported *block*, *cyclic*, and *block-cyclic* (sometimes called *cyclic*(k)) distribution of templates onto processors in multiple dimensions. For each distinct distribution, a different template needed to be specified. However, this mechanism could be used to ensure that arrays of different sizes, such as different meshes in multigrid calculations, would be mapped to the right processors.

Of course, a data-parallel language like Fortran D would not be useful unless it could be compiled to code with reasonable performance on each parallel target platform. The challenge was to determine how to decompose the computations and map them to the processors in a way that would minimize the cost of communication. In addition, the compiler would need to generate communication when it was necessary and optimize that communication so that data was moved between pairs of processors in large blocks rather than sequences of single words. This was critical because, at the time, most of the cost of any interprocessor communication operation was the time to deliver the first byte (latencies were very large but bandwidth was quite reasonable). Fortran D was targeted to communication libraries that supported two-sided protocols: to get data from one processor to another, the owning processor had to send it, while the processor needing the data had to receive it. This made communication generation complicated because the compiler had to determine where both sends and receives were to be executed on each processor.

To address the issue of computation partitioning, the Rice project defined a strategy called "owner-computes", which locates computations on processors near the data where it is stored. In particular, the first Rice compiler prototypes[4] used "left-hand-side owner-computes", which compiles each statement so that all computations are performed on the processors owning the computation outputs.

[4] Strictly speaking the Fortran D compilers, and most research implementations of data-parallel languages were source-to-source translators that generated SPMD implementation, such as Fortran plus message-passing calls as the "object code." In fact, one (minor) motivation for initiating the MPI standardization effort was to provide a machine-independent target for data-parallel compilers.

The first compilation paper, by Callahan and Kennedy, describing the Rice strategy appeared in the 1988 LCPC conference at Cornell and was included in a collection from that conference in the Journal of Supercomputing [18]. (Earlier in the same year, Zima, Bast and Gerndt published their paper on the SUPERB parallelization tool [100], which also used a distribution-based approach.) Although Callahan and Kennedy described the owner-computes strategy in rudimentary form, the optimizations of communication and computation were performed locally, using transformations adapted from traditional code optimization. When this approach proved ineffective, the Rice group switched to a strategy that compiled whole loop nests, one at a time. This work was described in a series of papers that also covered the prototype implementation of this new approach [54, 53, 94]. Although they presented results on fairly small programs, these papers demonstrated substantive performance improvements that established the viability of distribution-based compilation. At the heart of the compilation process is a conversion from the global array index space provided in Fortran D to a local index space on each processor. To perform this conversion the compiler would discover, for each loop, which iterations required no communication, which required that data be sent to another processor, and which required that data be received before any computation could be performed. A summary of this approach appears in Chapter 14 of the book by Allen and Kennedy [4].

One important issue had to be dealt with in order to generate correct code for Fortran D over an entire program: how to determine, at each point in the program, what distribution was associated with each data array. Since distributions were not themselves data objects, they could not be explicitly passed to subprograms; instead they were implicitly associated with data arrays passed as parameters. To generate code for a subprogram, the compiler would have to follow one of three approaches: (1) perform some kind of whole program analysis of distribution propagation, (2) rely on declarations by the programmer at each subroutine interface, or (3) dynamically determine the distributions at runtime by inspecting a descriptor for each distributed array. The Fortran D group decided that dynamic determination would be too slow and relying on the programmer would be impractical, particularly when libraries might be written in the absence of the calling program. As a result, the Fortran D compiler performed an interprocedural analysis of the propagation of data distributions, making it a whole-program compiler from the outset. This simplified the issues at procedure boundaries, but complicated the compiler structure in a way that would later prove unpalatable to the HPF standardization group.

Vienna Fortran In 1985, a group at Bonn University in Germany, led by Hans Zima, started the development of a new compilation system, called SUPERB, for a data-parallel language in the context of the German Suprenum supercomputing project [41]. SUPERB [100] took as input a Fortran 77 program and a specification of a generalized block data distribution, producing an equivalent explicitly parallel message-passing program for the Suprenum distributed-memory architecture using the owner-computes strategy. This approach, which originally was interpreted as performing a computation on the processor owning the left-hand side of an assignment statement (or, more generally, the target of the computation), was later generalized to the selection of *any* processor that would maximize the locality of the computation. This turned out to be one of the most important ideas of HPF. Gerndt's Ph.D. dissertation [40], completed in 1989, is the first work describing in full detail the program transformations required for such a translation; an overview of the compilation technology is presented by Zima and Chapman in [101].

SUPERB was not a language design project: it focused on compiler transformations, using an ad-hoc notation for data distribution. However, after Zima's group relocated to University of Vienna, they began working, in collaboration with Piyush Mehrotra of NASA ICASE, on a full specification of a high-level data distribution language in the context of Fortran.

This new language, called Vienna Fortran, provided the programmer with a facility to define arrays of virtual processors, and introduced distributions as mappings from multidimensional array index spaces to (sub)sets of processor spaces. Special emphasis was placed on support for irregular and adaptive programs: in addition to the regular *block* and *block-cyclic* distribution classes the language provided *general block* and *indirect* distributions. *General block* distributions, inherited from SUPERB, partition an array dimension into contiguous portions of arbitrary lengths that may be computed at run time. Such distributions, when used in conjunction with reordering, can efficiently represent partitioned irregular meshes. *Indirect* distributions present another mechanism for the support of irregular problems by allowing the specification of arbitrary mappings between array index sets and processors. Both the general block and indirect distributions were later incorporated in the HPF 2.0 specification, described in Section 5. Implementing either of these distributions is difficult because the actual mappings are not known until run time. Therefore, the compiler must generate a preprocessing step, sometimes called an *inspector* [26, 96], that determines at run time a communication schedule and balances the loads across the different processors.

A key component of the Vienna Fortran language specification was a proposal for user-defined distributions and alignments. Although this feature was only partially implemented, it motivated research in the area of distributed sparse matrix representations [96] and provided important ideas for a recent implementation of such concepts in the Chapel high productivity language [17]. The Vienna Fortran Compilation System extended the functionality of the SUPERB compiler to cover most of the language, while placing strong emphasis on the optimization of irregular algorithms. An overview of the compilation technology used in this system is presented in Benkner and Zima [12].

CM Fortran CM Fortran was the premier commercial implementation of a data-parallel language. The language was developed by a team led by Guy Steele at Thinking Machines Corporation (makers of the Connection Machines) and a group of implementers led by David Loveman and Robert Morgan of COMPASS, Inc., a small software company. The goal of CM Fortran was to support development of technical applications for the CM-2, a SIMD architecture introduced in 1987. The original programming model for the CM-2 and its forerunner, the CM-1, had been *Lisp because the initial market had focused on applications in artificial intelligence. However, a Fortran compiler was released in 1991 [91], with the goal of attracting science and engineering users.

CM Fortran adopted the array assignment and array arithmetic extensions that had been incorporated into Fortran 90, but it also included a feature that was deleted from the Fortran 90 standard at the last minute: the `FORALL` statement, which permitted a particularly simple loop-like specification for array assignments. Arrays that were used in these statements were classified as CM arrays and mapped to virtual processor (VP) sets, one array element per processor. VPs were in turn mapped to the actual processors of the CM-2 in regular patterns. Optional `ALIGN` and `LAYOUT` directives could modify this mapping. Unlike Fortran D, CM Fortran used compiler directives, entered as comments, to specify data layouts for the data arrays in the CM-2 SIMD processor array. Each array assignment was compiled into a sequence of SIMD instructions to compute execution masks, move data, and invoke the actual computation. This was a simple and effective strategy that was extended in the Thinking Machines "slicewise" compiler strategy to reduce redundant data movement and masks.

Programmability in CM Fortran was enhanced by the availability of a rich library of global computation and communication primitives known as CMSSL, which was developed under the leadership of Lennart Johnsson. In addition to powerful computational primitives, it included a number of global operations, such as sum reduction, and scatter/gather operations, that are used to convert data in irregular memory patterns to compact arrays and vice versa. Many of these routines were later incorporated into the HPF Library specification.

When Thinking Machines introduced a MIMD architecture, the CM-5, in 1993, it retained the CM Fortran language for the new architecture, showing the popularity and portability of the model.

The HPF Standardization Process In November of 1991 at Supercomputing '91 in Albuquerque, New Mexico, Kennedy and Fox were approached about the possibility of standardizing the syntax of data-parallel versions of Fortran. The driving forces behind this effort were the commercial vendors, particularly Thinking Machines, who were producing distributed-memory scalable parallel machines. Digital Equipment Corporation (DEC) was also interested in producing a cross-platform for Fortran that included many of the special features included in DEC Fortran.

In response to the initial discussions, Kennedy and Fox met with a number of academic and industrial representatives in a birds-of-a-feather session. The participants agreed to explore a more formal process through a group that came to be known as the High Performance Fortran Forum (HPFF). Kennedy agreed to serve as the HPFF chair and Charles Koelbel assumed the role of Executive Director. With support from the Center for Parallel Computation Research (CRPC) at Rice University, a meeting was hurriedly organized in Houston, Texas, in January 1992. Interest in the process was evident from the overflow attendance (nearly 100 people) and active discussions during presentations. The meeting concluded with a business session in which more than 20 companies committed to a process of drafting the new standard.

Because the interest level was high and the need pressing, it was agreed that the process should try to produce a result in approximately one year. Although we did not fully realize it at the time, this "management" agreement would affect the features of the language. The timeline forced us to use the informal rule that HPF would adopt only features that had been demonstrated in at least one language and compiler (including research projects' compilers). This of course limited some of the features considered, particularly in the realm of advanced data distributions. We have to admit, however, that the "at least one compiler" rule was, as Shakespeare might have said, oft honored in the breach. In particular, the committee felt that no fully satisfactory mechanism for subroutine interfaces had been demonstrated, and therefore fashioned a number of complementary features.

The active HPFF participants (30-40 people) met for two days every six weeks or so, most often in a hotel in Dallas, Texas, chosen because of convenience to airline connections.[5] There was a remarkable collection of talented individuals among the regular attendees. Besides Kennedy and Koelbel, those participants who were editors of the standard document included Marina Chen, Bob Knighten, David Loveman, Rob Schreiber, Marc Snir, Guy Steele, Joel Williamson, and Mary Zosel. Table 1 contains a more complete list of HFFF attendees, ordered by their affiliations at the time. We include the affiliations for two reasons: each organization (that had been represented at two of the past three meetings) had one vote, and the affiliations illustrate the breadth of the group. There are representatives from hardware and software vendors, university computer science researchers, and government and industrial application users. HPFF was a consensus-building process, not a single-organization project.

This group dealt with numerous difficult technical and political issues. Many of these arose because of the tension between a need for high language functionality and the desire for implementation simplicity. It was clear from the outset that HPF would need to be flexible and powerful enough to implement a broad range of applications with good performance. However, real applications differ significantly in the way they deal with data and in their patterns of computation, so different data distributions would be needed to accommodate them. Each data distribution built in to the language required a lot of effort from the compiler developers. So how many data distributions would be enough? As a case in point, consider the the *block-cyclic* distribution in which groups of *k* rows or columns of an array are assigned in round-robin fashion to the processors in a processor array. From the experience with the research compilers, the implementation of this distribution was unquestionably going to be challenging. Some members of the HPF Forum argued against it. In the end, however, it was included because it was needed to balance the computational load across processors on triangular calculations such as those done in dense linear algebra (e.g., LU Decomposition). In general, the need for *load balancing*, which attempts to assign equal amounts of computation to each processor for maximum speedup, motivated the need for a rich set of distribution patterns.

The problems of power versus complexity were compounded by limited experience with compilation of data-parallel languages. The research compilers were mostly academic prototypes that had been applied to few applications of any size. CM Fortran, on the other hand, was relatively new and was not as feature-rich as either Fortran D or Vienna Fortran. Therefore many decisions had to be made without full understanding of their impact on the complexity of the compilers.

Another difficult decision was whether to base the language on Fortran 77 or Fortran 90, the new standard as of the beginning of the HPF process. The problem was that most of the companies making parallel machines had compilers that handled Fortran 77 only. They were reluctant to commit to implementing the full Fortran 90 standard as a first step toward HPF. On the other hand, the companies that had already based their compilers on Fortran 90, such as Thinking Machines, wanted to take advantage of the language advances.

In the end, the users participating in the process tipped the balance toward Fortran 90. There were many reasons for this. Primary among them was that Fortran 90 included features that would make the HPF language definition cleaner, particularly at subprogram interfaces. In addition, Fortran 90

[5] In part by design, the same hotel was also used for meetings of the Message Passing Interface Forum (MPIF) which took place a bit later. One participant in both forums from England later joked that it was the only hotel in the world where he could order "the usual" and get the right drink.

Table 1. Attendees (and institutions at the time) in HPFF process, 1992-1993

David Reese (Alliant)	Jerrold Wagener (Amoco)
Rex Page (Amoco)	John Levesque (APR)
Rony Sawdayi (APR)	Gene Wagenbreth (APR)
Jean-Laurent Philippe (Archipel)	Joel Williamson (Convex Computer)
David Presberg (Cornell Theory Center)	Tom MacDonald (Cray Research)
Andy Meltzer (Cray Research)	David Loveman (Digital)
Siamak Hassanzadeh (Fujitsu America)	Ken Muira (Fujitsu America)
Hidetoshi Iwashita (Fujitsu Laboratories)	Clemens-August Thole (GMD)
Maureen Hoffert (Hewlett Packard)	Tin-Fook Ngai (Hewlett Packard)
Richard Schooler (Hewlett Packard)	Alan Adamson (IBM)
Randy Scarborough (IBM)	Marc Snir (IBM)
Kate Stewart (IBM)	Piyush Mehrotra (ICASE)
Bob Knighten (Intel)	Lev Dyadkin (Lahey Computer)
Richard Fuhler (Lahey Computer)	Thomas Lahey (Lahey Computer)
Matt Snyder (Lahey Computer)	Mary Zosel (Lawrence Livermore)
Ralph Brickner (Los Alamos)	Margaret Simmons (Los Alamos)
J. Ramanujam (Louisiana State)	Richard Swift (MasPar Computer)
James Cownie (Meiko)	Barry Keane (nCUBE)
Venkata Konda (nCUBE)	P. Sadayappan (Ohio State)
Robert Babb II (OGI)	Vince Schuster (Portland Group)
Robert Schreiber (RIACS)	Ken Kennedy (Rice)
Charles Koelbel (Rice)	Peter Highnam (Schlumberger)
Don Heller (Shell)	Min-You Wu (SUNY Buffalo)
Prakash Narayan (Sun)	Douglas Walls (Sun)
Alok Choudhary (Syracuse)	Tom Haupt (Syracuse)
Edwin Paalvast (TNO-TU Delft)	Henk Sips (TNO-TU Delft)
Jim Bailey (Thinking Machines)	Richard Shapiro (Thinking Machines)
Guy Steele (Thinking Machines)	Richard Shapiro (United Technologies)
Uwe Geuder (Stuttgart)	Bernhard Woerner (Stuttgart)
Roland Zink (Stuttgart)	John Merlin (Southampton)
Barbara Chapman (Vienna)	Hans Zima (Vienna)
Marina Chen (Yale)	Aloke Majumdar (Yale)

included an array operation syntax that was consistent with the concept of global arrays. Such operations would make it easier to identify global operations that could be carried out in parallel. In retrospect, this decision may have been a mistake, because the implementation of Fortran 90 added too much complexity to the task of producing a compiler in a reasonable time frame. As we point out in Section 4, this may have contributed to the slow development of HPF compilers and, hence, to the limited acceptance of the language.

Another major technical issue was how to handle distribution information in subroutines. Developers of technical applications in Fortran had come to rely on the ability to build on libraries of subroutines that were widely distributed by academic institutions and sold by commercial enterprises such as IMSL and NAG. If HPF was to be successful, it should be possible to develop libraries that worked correctly on array parameters with different distributions. Thus, it should be possible to determine within a subroutine what data distributions were associated with the arrays passed into that subroutine.

Fortran D had dealt with this issue by mandating an interprocedural compiler. The vendors in the HPF Forum were understandably unwilling to follow this route because most commercial compilers included no interprocedural analysis capabilities (leaving aside the question of whether library vendors would be willing to provide source in the first place). Thus the language design needed a way to declare distributions of parameters so that HPF subroutine libraries could be used with different distributions, without suffering enormous performance penalties. A key issue here was whether to redistribute arrays at subroutine boundaries or somehow "inherit" the distribution from the calling program. Redistribution adds data movement costs on entry and exit to a subroutine, but allows the routine to employ a distribution that is optimized to the underlying algorithm. Inheriting the caller's distribution avoids redistribution costs, but may have performance penalties due to a distribution that is inappropriate for the algorithm or the cost of dynamically interpreting the distribution that is passed in. This discussion involved the most complex technical issues in the standardization process and the parts of the standard dealing with these issues are the most obscure in the entire document.

This example illustrates one of the problems with the HPF standardization process and the tight schedule that was adopted. Although the members agreed at the outset only to include features that had been tried in some prototype research or commercial implementation, the group sometimes ignored this guideline, substituting intuition for experience.

The result of this work was a new language that was finalized in early 1993 [50, 49] and presented at the Supercomputing Conference in the fall of the same year. The specifics of the language are discussed in the next section.

An additional set of HPFF meetings were held during 1994 with the goals of (1) addressing needed corrections, clarifications, and interpretations of the existing standard and (2) considering new features required in extending HPF to additional functions. The organization (including place and frequency) of the meetings was as in the first series. Ken Kennedy again served as Chair, while Mary Zosel took on the role of Executive Director. The attendees were mostly the same as in the 1992-1993 meetings, with the notable additions of Ian Foster (Argonne National Laboratory) and Joel Saltz (then at the University of Maryland), who each led subgroups considering future issues. The corrections were incorporated in the High Performance Fortran 1.1 standard, a slight revision of the 1993 document presented at the Supercomputing '94 conference in Washington, DC in November 1994. The HPF 1.1 document itself refers to some of the issues needing clarification as "dark corners" of the language (those issues do not affect the examples in the next section). The new features and clarifications discussed but not included in HPF 1.1 were collected in the HPF Journal of Development and served as a starting point for the HPF 2 standardization effort, discussed in Section 5.

3. The HPF Language

The goals established for HPF were fairly straightforward:

- To provide convenient programming support for scalable parallel computer systems, with a particular emphasis on data parallelism
- To present an accessible, machine-independent programming model with three main qualities: (1) The application developer should be able to view memory as a single shared address space, even on distributed-memory machines; in other words, arrays should be globally accessible but distributed across the memories of the processors participating in a computation. (2) Programs written in the language should appear to have a single thread of control, so that the program could be executed correctly on a single processor; thus, all parallelism should derive from the parallel application of operations to distributed data structures. (3) Communication should be implicitly generated, so that the programmer need not be concerned with the details of specifying and managing inter-processor message passing.
- To produce code with performance comparable to the best hand-coded MPI for the same application.

To achieve these goals the HPF 1.0 Standard defined a language with a number of novel characteristics.

First, the language was based on Fortran 90, with the extensions defined as a set of "directives" in the form of Fortran 90 comments. These directives could be interpreted by HPF compilers as advice on how to produce a parallel program. On a scalar machine, an HPF program could be executed without change by simply ignoring the directives, assuming the machine had sufficient memory. This device, which was later copied by the OpenMP standards group, permitted the HPF parallelism extensions to be separated cleanly from the underlying Fortran 90 program: the program still needed to embody a data-parallel algorithm, but the same program should work on both sequential and parallel systems. Compilers for the sequential systems would not be required to recognize any HPF directives. It should be noted that the ability to run essentially the same program on both a sequential and parallel machine is a huge advantage for the application programmer in debugging an algorithm.

The principal additions to the language were a set of *distribution directives,* that specified how arrays were to be laid out across the memories of the machine. Sets of arrays could be aligned with one another and then distributed across the processors using built-in distributions, such as *block*, *cyclic*, and *block-cyclic*. These directives could be used to assign the individual rows or columns to processors in large blocks, or smaller blocks in round-robin fashion. We illustrate this by showing the application of directives to a simple relaxation loop:

```
REAL A(1000,1000), B(1000,1000)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
&          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

The `DISTRIBUTE` directive specifies how to partition a data array onto the memories of a real parallel machine. In this case, it is most natural to distribute the first dimension, since iterations over it can be performed in parallel. For example, the programmer can distribute data in contiguous chunks across the available processors by inserting the directive

```
!HPF$ DISTRIBUTE A(BLOCK,*)
```

after the declaration of `A`. HPF also provides other standard distribution patterns, including `CYCLIC` in which elements are assigned to processors in round-robin fashion, or `CYCLIC(K)` by which blocks of `K` elements are assigned round-robin to processors. Generally speaking, `BLOCK` is the preferred distribution for computations with nearest-neighbor elementwise communication, while the CYCLIC variants allow finer load balancing of some computations. Also, in many computations (including the example above), different data arrays should use the same or related data layouts. The ALIGN directive specifies an elementwise matching between arrays in these cases. For example, to give array B the same distribution as A, the programmer would use the directive

```
!HPF$ ALIGN B(I,J) WITH A(I,J)
```

Integer linear functions of the subscripts are also allowed in ALIGN and are useful for matching arrays of different shapes.

Using these directives, the HPF version of the example code is:

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
&          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

Once the data layouts have been defined, implicit parallelism is provided by the *owner-computes* rule, which specifies that calculations on distributed arrays should be assigned in such a way that each calculation is carried out on the processors that own the array elements involved in that calculation. Communication would be implicitly generated when a calculation involved elements from two different processors.

As the Fortran D project and Vienna Fortran projects showed, data distribution of subroutine arguments was a particularly complex area. To summarize the mechanism that HPF eventually adopted, formal subroutine arguments (i.e. the variables as declared in the subroutine) could have associated ALIGN and DISTRIBUTE directives. If those directives fully specified a data distribution, then the actual arguments (i.e. the objects passed by the subroutine caller) would be redistributed to this new layout when the call was made, and redistributed back to the original distribution on return. Of course, if the caller and callee distributions matched, it was expected that the compiler or runtime system would forego the copying needed in the redistribution. HPF also defined a system of "inherited" distributions by which the distribution of the formal arguments would be identical to the actual arguments. This declaration required an explicit subroutine interface, such as a Fortran 90 INTERFACE block. In this case, no copying would be necessary, but code generation for the subroutine would be much more complex to handle all possible incoming distributions. This complexity was so great, in fact, that to our knowledge no compiler fully implemented it.

In addition to the distribution directives, HPF has special directives that can be used to assist in the identification of parallelism. Because HPF is based on Fortran 90, it also has array operations to express elementwise parallelism directly. These operations are particularly appropriate when applied to a distributed dimension, in which case the compiler can (relatively) easily manage the synchronization and data movement together. Using array notation in this example produces the following:

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
    A(2:N,J) = &
&     (A(2:N,J+1)+2*A(2:N,J)+A(I,J-1))*0.25 &
&    + (B(3:N+1,J)+2*B(2:N,J)+B(1:N-1,J))*0.25
ENDDO
```

In addition to these features, HPF included the ability to specify that the iterations of a loop should be executed in parallel. Specifically, the INDEPENDENT directive says that the loop that follows is safe to execute in parallel. This can be illustrated with the code from the example above.

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  !HPF$ INDEPENDENT
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
&          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

Use of the directive ensures that a parallel loop will be generated by any HPF compiler to which the program is presented. Many compilers can detect this fact for themselves when analyzing programs with subscript expressions that are linear in the loop variables (as in the above example), based on dependence analysis along with the distribution information. However, the INDEPENDENT directive is essential for loops that are theoretically unanalyzable—for example, loops iterating over the edges of an unstructured mesh, which contain subscripted subscripts. Often the programmer will have application-specific knowledge that allows such loops to be executed in parallel.

Although the INDEPENDENT directive violates the goal of presenting a single thread of control, the issue was sidestepped in the standard by defining the directive as an assertion that the loop had no inter-iteration dependencies that would lead to data races; if this assertion was incorrect, the program was declared to be not *standard-conforming*. Thus, a standard-conforming HPF program would always produce the same answers on a scalar machine as a parallel one. Unfortunately, this feature made it impossible to determine at compile time whether a program was standard-conforming.

HPF also provided the FORALL statement, taken from CM Fortran and early drafts of Fortran 90, as an alternative means of expressing array assignment. The nested DO loop in our relaxation example could be written as

```
FORALL (J = 2:N, I=2:N) &
&   A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
&          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Semantically, the FORALL was identical to an array assignment; it computed the values on the right-hand side for all index values before storing the results into any left-hand side location. (There was also a multi-statement FORALL that applied this semantic rule to all assignments in the body in turn.) The explicit indexing allowed FORALL to conveniently express a wider range of array shapes and computations than the standard array assignment, as in the following example.

```
! Assignment to a diagonal, computed its index
FORALL (I=1:N) A(I,I) = I*I
```

High Performance Fortran was one of the first languages to include the specification for an associated library, the *HPF Library*, as a part of the defined language. Special global operations, such as sum reduction, gather and scatter, and partial prefix operations were provided by the HPF Library, which incorporated many parallel operations on global arrays that proved to be useful in other data-parallel languages, such as CM Fortran. This library added enormous power to the language. Specification of an associated library is now standard practice in C, C++, and Java.

Finally, HPF included a number of features that were designed to improve compatibility and facilitate interoperation with other programming languages and models. In particular, the EXTRINSIC interface made it possible to invoke subprograms that were written in other languages such as scalar Fortran and C. Of particular importance was the ability to call subroutines written in MPI in a way that made it possible to recode HPF subprograms for more efficiency.

4. Experience with the Language

The initial response to HPF could be characterized as cautious enthusiasm. A large part of the user community, those who had not already recoded using explicit message-passing, was hopeful that the language would permit a high-level programming interface for parallel machines that would make parallel programs portable and efficient without the need for extensive coding in MPI or its equivalent. The vendors, on the other hand, were hoping that HPF would expand the market for scalable parallel computing enough to increase profitability. Several vendors initiated independent compiler efforts, including Digital [46], IBM [43], and Thinking Machines. A number of other hardware vendors offered OEM versions of compilers produced by independent software vendors such as the Portland Group, Inc. (PGI) [15] and Applied Parallel Research [8]. At its peak, there were 17 vendors offering HPF products and over 35 major applications written in HPF, at least one of which was over 100,000 lines of code.

Nevertheless, as experience with the language increased, so did frustration on the part of the users. It became clear that it was not as easy as had been hoped to achieve high performance and portability in the language and many application developers gave up and switched to MPI. By the late 1990s usage of HPF in the United States had slowed to a trickle, although interest in Japan remained high, as we discuss below.

Given that HPF embodied a set of reasonable ideas on how to extend an existing language to incorporate data parallelism, why did it not achieve more success? In our view there were four main reasons: (1) inadequate compiler technology, combined with a lack of patience in the HPC community; (2) insufficient support for important features that would make the language suitable for a broad range of problems; (3) the inconsistency of implementations, which made it hard for a user to achieve portable performance; and (4) the complex relationship between program and performance, which made performance problems difficult to identify and eliminate. In the paragraphs that follow, we explore each of these issues in more detail.

Immature Compiler Technology When HPF was first released, Kennedy gave numerous addresses to technical audiences cautioning them to have limited expectations for the first HPF compiler releases. There were many reasons why caution was appropriate.

First, HPF was defined on top of Fortran 90, the first major upgrade to the Fortran standard since 1977. Furthermore, the Fortran 90 extensions were not simple: implementing them would require an enormous effort for the compiler writers. Among the new features added to Fortran 90 were (1) mechanisms for whole and partial array handling that required the use of descriptor-based implementations and "scalarization" of array assignments, (2) modules and interface blocks (which HPF depended on for specification of the distributions of arrays passed to subprograms), (3) recursion, and (4) dynamic storage allocation and pointer-based data structures. Since none of these features were present in Fortran 77, building a Fortran 90 compiler required a substantive reimplementation to introduce stack frames for recursion, heap storage, and array descriptors. Moving from Fortran 77 to Fortran 90 involved an effort comparable to building a compiler from scratch (except for the low-level code generation). At the time of the first HPF specification in 1993, most companies had not yet released their first Fortran 90 compilers. Thus, to produce an implementation of HPF, those companies would first need to implement almost all of Fortran 90, putting a huge obstacle in the way of getting to HPF.

Second, the entire collection of features in HPF, including the HPF library, was quite extensive and required new compilation strategies that, at the time of the release of HPF 1.0, had only been implemented in research compilers and the CM Fortran compiler. Proper compilation of HPF requires extensive global analysis of distributions, partitioning of computation, generation of communication, and optimizations such as overlapping communication and computation. Implementing all of these well would require compilers to mature over a number of years, even if implementing Fortran 90 were not a precondition.

Finally, efficient implementation of HPF programs required that the compiler pay special attention to locality on individual processors. Since most of the processors used in distributed-memory systems were uniprocessors with complex cache hierarchies, generating efficient code required that advanced transformation strategies, such as tiling for cache reuse, be employed. At the time of the release of the initial HPF specification, these techniques were beginning to be understood [39, 98, 64], but most commercial compilers had not yet incorporated them [84, 28].

In spite of this, the HPC community was impatient for a high-level programming model, so there was heavy pressure on the compiler vendors to release some implementation of HPF. As a result the first compilers were premature, and the performance improvements they provided were disappointing in all but the simplest cases.

Meanwhile, the developers of parallel applications had deadlines to meet: they could not afford to wait for HPF to mature. Instead they turned to Fortran with explicit MPI calls, a programming model that was complex but was, at least, ready to use. The migration to MPI significantly reduced the demand for HPF, leading compiler vendors to reduce, or even abandon, the development effort. The end result was that HPF never achieved a sufficient level of acceptance within the leading-edge parallel computing users to make it a success. In a very real sense, HPF missed the first wave of application developers and was basically dead (or at least considered a failure) before there was a second wave.

Missing Features To achieve high performance on a variety of applications and algorithms, a parallel programming model must support a variety of different kinds of data distributions. This is particularly critical for sparse data structures or adaptive algorithms. The original specification of HPF included only three main distributions: `BLOCK`, `CYCLIC`, and `CYCLIC(K)`. These distributions are effective for dense array computations and, in the case of `CYCLIC(K)`, even linear algebra. However, many important algorithmic strategies were difficult, or even impossible, to express within HPF without a big sacrifice of performance. This deficiency unnecessarily narrowed the space of applications that could be effectively expressed in HPF. As a result, the developers of applications requiring distributions that were not supported went directly to MPI. Although this problem was partially corrected in HPF 2.0 (discussed later), the damage was already done. In the final section of this paper, we suggest a strategy for addressing this problem in future data-parallel languages.

A second issue with HPF was its limited support for task parallelism. The parallel loop feature helped a little, but users wanted more powerful strategies for task parallelism. Once again, this was corrected in HPF 2.0, but it was too late. What should have happened was a merger between the features in HPF and OpenMP, discussed below.

Barriers to Achieving Portable Performance One of the key goals of HPF was to make it possible for an end user to have one version of a parallel program that would then produce implementations on different architectures that would achieve a significant fraction of the performance possible on each architecture. This was not possible for two main reasons.

First, different vendors focused on different optimizations in their HPF implementations. This caused a single HPF application to achieve dramatically different performance on the machines from different vendors. In turn, this led users to recode the application for each new machine to take advantage of the strengths (and avoid the weaknesses) of each vendor's implementation, thwarting the original goal of absolute portability.

Second, the HPF Library could have been used to address some of the usability and performance problems described in previous sections. For example, the "gather" and "scatter" primitives could have been used to implement sparse array calculations. However, there was no open-source reference implementation for the library, so it was left to each compiler project to implement its own version. Because of the number and complexity of the library components, this was a significant implementation burden. The end result was that too little attention was paid to the library and the implementations were inconsistent and exhibited generally poor performance. Thus, users were once again forced to code differently for different target machines, using those library routines that provided the best performance.

Difficulty of Performance Tuning Every HPF compiler we are aware of translated the HPF source to Fortran plus MPI. In the process, many dramatic transformations were carried out, making the relationship between what the developer wrote and what the parallel machine executed somewhat murky. This made it difficult for the user to identify and correct performance problems.

To address the identification problem, the implementation group at Rice collaborated with Dan Reed's group at the University of Illinois to map the Pablo Performance Analysis Systems diagnostics, which were based on a message-passing architecture, back to HPF source [2]. This effort was extremely effective and was implemented in at least one commercial compiler, but it did little to address the tuning problem. That is, the user could well understand what was causing his or her performance problem, but have no idea how to change the HPF source to overcome the issue. Of course, he or she could use the EXTRINSIC interface to drop into MPI, but that voided the advantages of using HPF in the first place.

The identification problem in the context of performance tuning was also addressed in a cooperation between the Vienna Fortran group and Maria Calzarossa's research group at the University of Pavia, Italy. This project developed a graphical interface in which the explicitly parallel MPI-based target code, with performance information delivered by the MEDEA tool, could be linked back to the associated source statements. For example, a programmer using this tool and familiar with the code generated for an independent loop in the framework of the *inspector/executor* paradigm [83] was able to analyze whether the source of a performance problem was the time required for the distribution of work, the (automatic) generation of the communication schedule by the inspector, or the actual communication generated for the loop.

5. The HPF 2 Standardization Effort

In an attempt to correct some of the deficiencies in HPF 1.0 and 1.1, the HPF Forum undertook a second standardization effort from 1995 to 1996. This effort led to the HPF 2.0 standard which incorporated a number of new features:

1. The REDUCTION clause for INDEPENDENT loops, substantially expanding the cases where INDEPENDENT could be used. (HPF 1.0 had the NEW clause for INDEPENDENT to allow "local" variables to each loop iteration, but no straightforward way to allow simple accumulations.)
2. The new HPF_LIBRARY procedures SORT_DOWN, SORT_UP, to perform sorting. (HPF 1.0 had already introduced the GRADE_UP and GRADE_DOWN functions, which produced permutation vectors rather than sorting elements directly.)
3. Extended data mapping capabilities, including mapping of objects to *processor subsets*; mapping of pointers and components of derived types; the GEN_BLOCK and INDIRECT distribution patterns and the RANGE and SHADOW modifiers to distributions. (HPF 1.0, as noted above, was limited to very regular distribution patterns on array variables.)
4. Extended parallel execution control, including the ON directive to specify the processor to execute a computation, the RESIDENT directive to mark communication-free computations, and the TASK_REGION directive providing coarse-grain parallel tasks. (HPF 1.0 left all computation mapping to the compiler and runtime system.)
5. A variety of additional intrinsic and HPF_LIBRARY procedures, mostly concerned with querying and managing data distributions. (HPF 1.0 had some support, but more was found necessary.)
6. Support for asynchronous I/O with a new statement WAIT, and an additional I/O control parameter in the Fortran READ/WRITE statement. (HPF 1.0 had ignored I/O facilities.)

Except for the first two items above, these new features were "Approved Extensions" rather than the "Core Language". HPFF had intended that vendors would implement all of the core language features (e.g. the REDUCTION clause) immediately, and prioritize the extensions based on customer demand. Not surprisingly, this created confusion as the feature sets offered by different vendors diverged. To our knowledge, no commercial vendor or research project ever attempted an implementation of the full set of approved extensions.

6. The Impact and Influence of HPF

Although HPF has not been an unqualified success, its has been enormously influential in the development of high-level parallel languages. The current CiteSeer database lists 827 citations for the original 1993 technical report, which was later published in Scientific Programming [50], making it the 21st most cited document in the computer science field (as covered by CiteSeer). In addition, over 1500 publications in CiteSeer refer to the phrase "High Performance Fortran". Many of these papers present various approaches to implementing the language or improving upon it, indicating that it generated a great deal of intellectual activity in the academic community.

6.1 Impact on Fortran and its Variants

Fortran 95 While the meetings that led to HPF 1.1 were underway, the X3J3 committee of ANSI was also meeting to develop the Fortran 95 standard [1]. That group had long watched developments in HPF with an eye toward adopting successful features, with Jerry Wagener serving as an informal liaison between the groups. Ken Kennedy gave a presentation to X3J3 on HPF's parallel features, and ensured that no copyright issues would hamper incorporation of HPF features into the official Fortran standard. When the Fortran 95 standard [1] was officially adopted in 1996, it included the HPF FORALL and PURE features nearly verbatim. The new standard also included minor extensions to MAXLOC and MINLOC that had been adopted into the HPF Library from CM Fortran. Both HPFF and X3J3 considered this sharing a positive development.

HPF/JA In 1999, the Japan Association for High Performance Fortran, a consortium of Japanese companies including Fujitsu, Hitachi, and NEC, released HPF/JA [86], which included a number of features found in previous programming languages on parallel-vector machines from Hitachi and NEC. An important source contributing to HPF/JA was the HPF+ language [11] developed and implemented in a European project led by the Vienna group, with NEC as one of the project partners. HPF+, resulting from an analysis of advanced industrial codes, provided a `REUSE` clause for independent loops that asserted reusability of the communication schedule computed during the first execution of the loop. In the same context, the `HALO` construct of HPF+ allowed the functional specification of nonlocal data accesses in processors and user control of the copying of such data to region boundaries.

These and other features allowed for better control over locality in HPF/JA programs. For example, the `LOCAL` directive could be used to specify that communication was not needed for data access in some situations where a compiler would find this fact difficult to discern. In addition, the `REFLECT` directive included in HPF/JA corresponds to HPF+'s `HALO` feature. HPF/JA was implemented on the Japanese Earth Simulator [86], discussed below.

OpenMP Finally, we comment on the relationship between HPF and OpenMP [73, 33]. OpenMP was proposed as an extension of Fortran, C, and C++, providing a set of directives that support a well-established portable shared memory programming interface for SMPs based on a fork/join model of parallel computation. It extends earlier work performed by the Parallel Computing Forum (PCF) as part of the X3H5 standardization committee [66] and the SGI directives for shared memory programming. OpenMP followed the HPF model of specifying all features in the form of directives, which could be ignored by uniprocessor compilers. Most of the OpenMP features were completely compatible with HPF: in fact, the parallel loop constructs were basically extensions of the HPF parallel loop construct.

OpenMP is an explicitly parallel programming model in which the user is able to generate threads to utilize the processors of a shared memory machine and is also able to control accesses to shared data in an efficient manner. Yet, the OpenMP model does not provide features for expressing the mapping of data to processors in a distributed-memory system, nor does it permit the specification of processor/thread affinity. Such a paradigm will work effectively as long as locality is of no concern.

This shortcoming was recognized early, and a number of attempts have been made to integrate OpenMP with language features that allow locality-aware programming [72, 13]. However, the current language standard does not support any of these proposals. In fact, the recent development of *High Productivity Languages* (see Section 6.2) may render such attempts obsolete, since all these languages integrate multithreading and locality awareness in a clean way.

As a final remark, it is interesting to note that the OpenMP design group, which started its work after the release of HPF 1.0, defined syntax for its directives that was incompatible with HPF syntax. (From the OpenMP side, it was probably equally puzzling that HPF had defined its own syntax rather than adopt the existing PCF constructs.) The language leaders of both efforts met on several occasions to explore the possibility of unifying the languages, as this would have provided needed functionality to each, but a combined standardization project never got started.

HPF Usage After the initial release of High Performance Fortran, there were three meetings of the HPF Users Group: one in Santa Fe, NM (February 24-26, 1997), the second in Porto, Portugal (June 25-26, 1998) and most recently in Tokyo, Japan (October 18-20, 2000). At these meetings surveys were taken to determine usage and product development. The papers presented at the Tokyo meeting were collected into a special double issue of *Concurrency, Practice and Experience* (Volume 14, Number 8-9, August 2002). The Tokyo meeting was remarkable in demonstrating the continuing high interest in HPF within Japan. This was reemphasized later when the Japanese Earth Simulator [86] was installed and began running applications. The Earth Simulator featured a high-functionality HPF implementation, initially based on the Portland Group compiler, that supported the HPF/JA extensions. Two applications described in the CPE special issue were brought up on the Earth Simulator and achieved performance in the range of 10 Teraflops and above. The application Impact3D ran at nearly 15 Teraflops, or roughly 40 percent of the peak speed of the machine [82]; it was awarded a Gordon Bell Prize at SC2002 for this achievement. Neither of these applications included any MPI.

6.2 HPCS Languages

Although HPF missed the leading edge of parallel application developers, its ideas are finding their way into newer programming languages. Recently, DARPA has funded three commercial vendors to develop prototype hardware and software systems for the High Productivity Computing Systems (HPCS) projects. A principal goal of this effort is to ease the burden of programming leading-edge systems that are based on innovative new architectures. The three vendors have produced three new language proposals: Chapel (Cray) [17, 20, 36], Fortress (Sun) [90], and X10 (IBM) [22]. All of these include data parallelism in some form.

Chapel, Fortress, and X10 are new object-oriented languages supporting a wide range of features for programmability, parallelism, and safety. They all provide a global name space, explicit multithreading, and explicit mechanisms for dealing with locality. This includes features for distributing arrays across the computing nodes of a system, and establishing affinity between threads and the data they are operating upon. Finally, these languages support both data and task parallelism. Because the HPCS language designers were familiar with the HPF experience, they have constructed data-parallel features that address many of the shortcomings described in Section 4.

In contrast to the directive-oriented approach of HPF or the library-based specification of MPI, Chapel, X10, and Fortress are *new* memory-managed object-oriented languages supporting a wide range of safety features. They build upon the experiences with object-oriented languages and their implementation technology over the past decade, but try to eliminate well-known shortcomings, in particular with respect to performance. At the same time, they integrate key ideas from many parallel languages and systems, including HPF.

X10 is based on Java; however, Java's support for concurrency and arrays has been replaced with features more appropriate for high performance computing such as a partitioned global address space. In contrast, Chapel is a completely new language based on what are perceived as the most promising ideas in a variety of current object-oriented approaches.

Locality Management All three languages provide the user with access to virtual units of locality, called respectively *locales* in Chapel, *regions* in Fortress, or *places* in X10. Each execution of a program is bound to a set of such locality units, which are mapped by the operating system to physical entities, such as computational nodes. This provides the user with a mechanism to (1) distribute data collections across locality units, (2) align different collections of data, and (3) establish affinity between computational threads and the data they operate upon. This approach represents an obvious generalization of key elements in HPF.

Fortress and X10 provide extensive libraries of built-in distributions, with the ability to produce new user-specified data distributions by decomposing the index space or combining distributions in different dimensions. In Fortress, all arrays are distributed by default; if no distribution is specified, then an array is assigned the default distribution. Chapel, on the other hand, has no built-in distributions but provides an extensive framework supporting the specification of arbitrary user-defined distributions that is powerful enough to deal with sparse data representations [36]. X10 has a *Locality Rule* that disallows the direct read/write of a remote reference in an activity. Chapel and Fortress do not distinguish in the source code between local and remote references.

Multithreading HPF specifies parallel loops using the *independent* attribute, which asserts that the loop does not contain any loop-carried dependences, thus excluding data races. Chapel distinguishes between a sequential *for* loop and a parallel *forall* loop, which iterates over the elements of an index domain, without a restriction similar to HPF's *independent* attribute. Thus the user is responsible for avoiding dependences that lead to data races.

The Fortress for-loop is parallel by default, so if a loop iterates over a distributed dimension of an array the iterations will be grouped onto processors according to the distributions. A special "sequential" distribution can be used to serialize a for-loop. The Fortress compiler and run-time system are free to rearrange the execution to improve performance so long as the meaning of the program under the distribution-based looping semantics is preserved. In particular, the subdivision into distributions essentially overpartitions the index space so that operations can be dynamically moved to free processors to optimize load balance.

X10 distinguishes two kinds of parallel loops: the *foreach* loop, which is restricted to a single locality unit, and the *ateach* loop that allows iteration over multiple locality units. As with the *Locality Rule* discussed above, X10 pursues a more conservative strategy than Chapel or Fortress, forcing the programmer to distinguish between these two cases.

6.3 Parallel Scripting Languages

Although the Fortran and C communities were willing to tolerate the difficulties of writing MPI code for scalable parallel machines, it seems unlikely that the large group of users of high-level scripting languages such as Matlab, R, and Python will be willing to do the same. Part of the reason for the popularity of these languages is their simplicity.

Nevertheless, there is substantive interest in being able to write parallel code in these languages. As a result, a number of research projects and commercial endeavors, including The MathWorks, have been exploring strategies for parallel programming, particularly in Matlab [76, 35, 55, 61]. Most of these projects replace the standard Matlab array representation with a global distributed array and provide replacements for all standard operators that perform distributed operations on these arrays. Although this is in the spirit of HPF, the overhead of producing operation and communication libraries by hand limits the number of different distributions that can be supported by such systems. Most of the current implementations are therefore restricted to the distributions that are supported by ScaLAPACK, the parallel version of LAPACK, which is used to implement array operations in the Matlab product.

A recently initiated project at Rice, led by Kennedy, is seeking to build on the Rice HPF compiler technology, described in Section 7 below, to provide a much richer collection of array distributions. The basic idea behind this effort is to produce the distributed array operations in HPF and allow the HPF compiler to specialize these library routines to all the different supported distributions. Currently, this list includes sequential(*), block, block-cyclic, and block(k), in each dimension, plus a new, two-dimensional distribution called "multipartitioning" that is useful for achieving the best possible performance on the NAS Parallel Benchmarks. However, a number of new distributions are being contemplated for the near future.

The goal of these efforts is to provide the scripting language community, and especially Matlab users, with a simple way to get reasonably scalable parallelism with only minimal change to their programs. If they are successful, they will not only vindicate the vision behind HPF, but will also dramatically increase the community of application developers for scalable parallel machines.

7. Lessons Learned

For the future, there are some important lessons to be learned from the HPF experience: in particular, what should be done differently in future languages with similar features. In this section, we discuss a few of these issues.

First, we have learned a great deal about compiler technology in the twelve years since HPF 1.0 was released. It is clear that, on the first compilers, high performance was difficult to achieve in HPF. In fact, it became common practice to recode application benchmarks to exploit strengths of the HPF compiler for a particular target machine. In other words, application developers would rewrite the application for each new language platform. A case in point was the NAS parallel benchmark suite. A set of HPF programs that were coded by engineers at the Portland group was included as the HPF version of the NAS benchmarks. These versions were designed to get the highest possible performance on the Portland Group compiler, avoiding pitfalls that would compromise the efficiency of the generated code. Unfortunately, this compromised the HPF goal of making it possible to code the application in a machine-independent form and compile it without change to different platforms. In other words, the practice of coding to the language processor undermined HPF's ease of use, making it time consuming to port from one platform to another.

Over the past decade a research project at Rice University led by John Mellor-Crummey has been focused on the goal of achieving high performance on HPF programs that are minimally changed from the underlying Fortran 90. In the case of the NAS Parallel Benchmarks, this would mean that the sequential Fortran 90 program (embodying a potential parallel algorithm) would be changed only by the addition of appropriate distribution directives: the compiler should do the rest. The ultimate goal would be to produce code that was competitive with the hand-coded MPI versions of the programs that were developed for the NAS suite.

The HPF compiler produced by the Rice project, dHPF, was not able to achieve this goal because the MPI versions used a distribution, called *multipartitioning*, that was not supported in either HPF 1.0 or 2.0. When the Rice researchers added multipartitioning to the distributions supported by dHPF, they were able to achieve performance that was within a few percentage points of the hand-coded MPI versions for both the NAS SP and BT applications [25]. This result is confirmed by experience with HPF on the Earth Simulator and a subsequent experiment in which the dHFP compiler translated Impact3D to run on the Pittsburgh Supercomputer Center's Lemieux system (based on the Alpha processor plus a Quadrics switch) and achieved over 17 percent efficiency on processor counts from 128 to 1024 (the latter translated to 352 Gigaflops, likely a U.S. land speed record for HPF programs) [24].

The experience with multipartitioning illustrates another important aspect of making data-parallel languages more broadly applicable: there needs to be some way to expand the number of distributions available to the end user. Unfortunately, multipartitioning is but one of many possible distributions that one might want to add to an HPF compiler; the HPF 2.0 generalized block and indirect distributions are two others. There may be many other distributions that could be used to achieve high performance with a particular application on a particular parallel system. The base system cannot possibly include every useful distribution. This suggests the need for some mechanism for adding user-defined distributions. If we were able to define an interface for distributions and then define compiler optimizations in terms of that interface, we would have succeeded in separating data structure from data distribution. The end result would be a marked increase in the flexibility of the language. Research in this area is currently performed in the context of the Chapel programming language developed in the Cascade project. Chapel [17] provides no built-in distributions but offers a distribution class interface allowing the explicit specification of mappings, the definition of sequential and parallel iterators, and, if necessary, the control of the representation of distributions and local data structures.

As we indicated earlier, the differences in implementation strengths of the base language, combined with the paucity of good implementations of the HPF Library, were another reason for the limited success of the language. There were two dimensions to this problem. First, some companies never implemented the library at all. Others chose to spend their resources on specific routines that were needed by their customers and provide less than optimally efficient versions of the rest. The result was that the end user could not rely on the HPF Library if efficiency were a concern, which reduced the usability of the language overall.

One of the reasons for the success of MPI was the existence of a pretty good portable reference implementation called MPICH. A similar portable reference implementation for both the HPF compiler and the HPF Library would have given a significant boost to the language. The Portland Group compiler came pretty close to a reference implementation for the languages, but there was no corresponding standardized implementation for the library. Several leaders of the HPF implementer community discussed the possibility of securing Federal support for a project to provide a reference implementation of the HPF library but the funding was never found. This was particularly frustrating because there existed a very good implementation of most of the needed functionality in CMSSL (Connection Machine Scientific Subroutine Library), part of the Thinking Machines CMFortran run-time system. When Thinking Machines went out of business and was sold to Sun Microsystems, it might have been possible to spin out the needed library functionality had there been government support, but this second opportunity was also missed.

Performance tuning tools for HPF presented another set of problems. As indicated in our earlier discussion, a collaboration between compiler and tool developers could succeed in mapping performance information back to HPF source: the Pablo system did this with the Rice dHPF compiler and the Portland group compiler [2]. The difficulty was that there were only limited ways for a user to exercise fine-grained control over the code generated once the source of performance bottlenecks was identified, other than using the `EXTRINSIC` interface to drop into MPI. The HPF/JA extensions ameliorated this a bit by providing more control over locality. However, it is clear that additional features are needed in the language design to override the compiler actions where that is necessary. Otherwise, the user is relegated to solving a complicated inverse problem in which he or she makes small changes to the distribution and loop structure in hopes of tricking the compiler into doing what is needed.

As a final note, we should comment on the lack of patience of the user community. It is true that High Performance Fortran was rushed into production prematurely, but had people been willing to keep the pressure up for improvement rather than simply bolting to MPI, we could have persevered in providing at least one higher-level programming model for parallel computing. As the work at Rice and in Japan has shown, HPF can deliver high performance with the right compiler technology, particularly when features are provided that give the end user more control over performance. Unfortunately, the community in the United States was unwilling to wait. The United States federal government could have helped stimulate more patience by funding one or more of the grand-challenge application projects to use new tools like HPF, rather than just getting the application to run as fast as possible on a parallel platform. Because of this lack of patience and support, the second wave of parallel computing customers is unable to benefit from the experiences of the first. This is a lost opportunity for the HPC community.

8. Conclusion

High Performance Fortran failed to achieve the success we all hoped for it. The reasons for this were several: immature compiler technology leading to poor performance, lack of flexible distributions, inconsistent implementations, missing tools, and lack of patience by the community. Nevertheless, HPF incorporated a number of important ideas that will be a part of the next generation of high performance computing languages. These include a single-threaded execution model with a global address space, interoperability with other language models, and an extensive library of primitive operations for parallel computing. In addition, a decade of research and development has overcome many of the implementation impediments. Perhaps the time is right for the HPC community to once again embrace new data-parallel programming models similar to the one supported by HPF.

References

[1] Jeanne C. Adams, Walter S. Brainard, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 95 Handbook. Complete ISO/ANSI Reference*. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, 1997.

[2] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, November 1995.

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

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, California, 2002.

[5] Alliant Computer Systems. `http://en.wikipedia.org/wiki/Alliant-Computer-Systems`.

[6] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 112–125, New York, NY, USA, 1993. ACM Press.

[7] F. Andre, J.-L. Pazat, and H. Thomas. PANDORE: A System to Manage Data Distribution. In *International Conference on Supercomputing*, pages 380–388, Amsterdam, The Netherlands, June 1990.

[8] Applied Parallel Research, Sacramento, California. *Forge High Performance Fortran xhpf User's Guide, Version 2.1*, 1995.

[9] G. H. Barnes, R. M. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17:746–757, 1968.

[10] Kenneth E. Batcher. The Multi-Dimensional Access Memory in STARAN. *IEEE Transactions on Computers*, C-26(2):174–177, February 1977.

[11] S. Benkner, G. Lonsdale, and H.P. Zima. The HPF+ Project: Supporting HPF for Advanced Industrial Applications. In *Proceedings EuroPar'99 Parallel Processing*, volume 1685 of *Lecture Notes in Computer Science*. Springer–Verlag, 1999.

[12] S. Benkner and H. Zima. Compiling High Performance Fortran for Distributed-Memory Architectures. *Parallel Computing*, 1999.

[13] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C. Ofner. Extending OpenMP for NUMA Machines. In *Proceedings of Supercomputing 2000*, November 2000.

[14] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing '93*, November 1993.

[15] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF—An Optimizing High Performance Fortran Compiler for Distributed Memory Machines. *Scientific Programming*, 6(1):29–40, 1997.

[16] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986.

[17] D. Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60, April 2004.

[18] D. Callahan and K. Kennedy. Compiling Programs for Distributed–Memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[19] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.

[20] Bradford L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 2007. Special Issue on High Productivity Programming Languages and Models (in print).

[21] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[22] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[23] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–28, New York, NY, USA, 1993. ACM Press.

[24] Daniel Chavarría-Miranda, Guohua Jin, and John Mellor-Crummey. Assessing the US Supercomputing Strategy: An Application Study Using IMPACT-3D. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005.

[25] Daniel Chavarría-Miranda and John Mellor-Crummey. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. *The Journal of Instruction-Level Parallelism*, 5, February 2003. (http://www.jilp.org/vol5). Special issue with selected papers from: The Eleventh International Conference on Parallel Architectures and Compilation Techniques, September 2002. Guest Editors: Erik Altman and Sally McKee.

[26] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software Support for Irregular and Loosely Synchronous Problems. *International Journal of Computing Systems in Engineering*, 3(2):43–52, 1993. (also available as CRPC-TR92258).

[27] Peter Christy. Software to Support Massively Parallel Computing on the MasPar MP-1. In *Proceedings of the IEEE Compcon*, 1990.

[28] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.

[29] K. D. Cooper, M. W. Hall, K. Kennedy, and L. Torczon. Interprocedural Analysis and Optimization. *Communications in Pure and Applied Mathematics*, 48:947–1003, 1995.

[30] Cray history. `http://www.cray.com/about-cray/history.html`.

[31] Cray-1 Computer System. Hardware Reference Manual 224004. Technical report, Cray Research, Inc., November 1977. Revision C.

[32] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufman Publishers, San Francisco, California, 1999.

[33] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering*, 5(1):46–55, 1998.

[34] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. VM/EPEX—A VM Environment for Parallel Execution. Technical Report RC 11225(#49161), IBM T. J. & Watson Research Center, Yorktown Heights, New York, January 1985.

[35] Luiz DeRose and David Padua. A MATLAB to Fortran 90 Translator and Its Effectiveness. In *Proceedings of the 10th International Conference on Supercomputing*, May 1996.

[36] R. L. Diaconescu and H.P. Zima. An Approach to Data Distributions in Chapel. *The International Journal of High Performance Computing Applications*, 2006. Special Issue on High Productivity Programming Languages and Models (in print).

[37] M. Flynn. Some Computer Organisations and their Effectiveness. *Trans. Computers*, 21:948–960, 1972.

[38] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. The Fortran D Language Specification. Technical Report TR90–141, Department of Computer Science, December 1990.

[39] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *J. Parallel Distrib. Comput.*,

5(5):587–616, 1988.

[40] Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, Germany, 1989.

[41] Wolfgang K. Giloi. SUPRENUM: A Trendsetter in Modern Supercomputer Development. *Parallel Computing*, pages 283–296, 1988.

[42] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitsberg, W. Saphir, and M. Snir. *MPI—The Complete Reference: Volume 2, The MPI Extensions*. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, September 1998.

[43] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF Compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995.

[44] Manish Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[45] L. Hamel, P. Hatcher, and M. Quinn. An Optimizing C* Compiler for a Hypercube Multicomputer. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run–Time Environments for Distributed Memory Machines*. North–Holland, Amsterdam, The Netherlands, 1992.

[46] J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for Distributed–Memory Systems. *Digital Technical Journal of Digital Equipment Corporation*, 7(3):5–23, Fall 1995.

[47] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A Production Quality C* Compiler for Hypercube Machines. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, April 1991.

[48] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufman Publishers, 1996. Second Edition.

[49] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1–2):1–170, 1993. (also available as CRPC–TR92225).

[50] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Center for Research on Parallel Computation, Houston, Tex., 1993.

[51] R. Hill. MIMDizer: A New Tool for Parallelization. *Supercomputing Review*, 3(4), April 1990.

[52] W. Daniel Hillis. *The Connection Machine*. Series in Artificial Inteligence. MIT Press, Cambridge, MA, 1985.

[53] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed–Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[54] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed–Memory Machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[55] P. Husbands, C. Isbell, and A. Edelman. MATLAB*P: A Tool for Interactive Supercomputing. In *Proceedings of the 9th SIAM Conference on Parallel Processing*. Scientific Computing, 1999.

[56] ESA/390 Principles of Operation. Technical report, IBM Corporation. IBM Publication No. SA22-7201.

[57] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. In *Fifth Distributed-Memory Computing Conference*, pages 1105–1114, Charleston, South Carolina, April 1990.

[58] H. F. Jordan. The Force. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[59] Kendall Square Systems. `http://en.wikipedia.org/wiki/Kendall-Square-Research`.

[60] Ken Kennedy and Uli Kremer. Automatic Data Layout for Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, July 1998.

[61] Jeremy Kepner and Nadya Travinin. Parallel Matlab: The next generation. In *Proceedings of the 2003 Workshop on High Performance Embedded Computing (HPEC03)*, 2003.

[62] KSR1 Principles of Operation, 1991. Waltham, MA.

[63] D. Kuck, Y. Muraoka, and S. Chen. On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[64] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[65] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[66] B. Leasure. Parallel Processing Model for High-Level Programming Languages. Technical report, American National Standard for Information Processing, April 1994.

[67] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):361–376, 1991.

[68] J. Li and M.Chen. Generating Explicit Communication from Shared-Memory Program References. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.

[69] J. H. Merlin. Adapting Fortran 90 Array Programs for Distributed-Memory Architectures. In H.P. Zima, editor, *First International ACPC Conference*, pages 184–200. Lecture Notes in Computer Science 591, Springer Verlag, Salzburg, Austria, 1991.

[70] Message Passing Interface Forum. MPI: A Message–Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):165–414, 1994. (special issue on MPI, also available electronically via `ftp://www.netlib.org/mpi/mpi-report.ps`).

[71] Myrias Systems. `http://en.allexperts.com/e/m/my/myrias-research-corporation.htm`.

[72] D. S. Nikolopoulos, T.S. Papatheodoru, C.D. Polychronopoulos, J. Labarta, and E. Ayguade. Is Data Distribution Necessary in OpenMP? In *Proceedings of Supercomputing 2000*, November 2000.

[73] OpenMP Application Program Interface. Version 2.5. Technical report, OpenMP Architecture Review Board, May 2005. http://www.openmp.org.

[74] Anita Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[75] D. Pase. *MPP Fortran Programming Model*. High Performance Fortran Forum, January 1991.

[76] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary Results from a Parallel MATLAB Compiler. In *Proceedings of the International Parallel Processing Symposium*, pages 81–87, April 1998.

[77] A. P. Reeves and C.M. Chase. The Paragon Programming Paradigm and Distributed-Memory Multicomputers. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.

[78] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[79] John R. Rose and Guy L. Steele. C*: An Extended C Language for Data Parallel Programming. In *Proceedings Second International Conference on Supercomputing*, volume Vol.II,2-16, 1987. International Supercomputing Institute.

[80] M. Rosing, R.W. Schnabel, and R.P. Weaver. Expressing Complex Parallel Algorithms in DINO. In *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[81] R. Ruehl and M. Annaratone. Parallelization of Fortran Code on Distributed-Memory Parallel Processors. In *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990. ACM Press.

[82] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS Three-dimensional Fluid Simulation for Fusion Science with HPF on the Earth Simulator. In *SC2002*, November 2002.

[83] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time Scheduling and Execution of Loops on Message-passing Machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[84] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.

[85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[86] M. Shimasaki and Hans P. Zima. Special Issue on the Earth Simulator, November 2004.

[87] D. Slotnick, W. Brock, and R. MacReynolds. The SOLOMON computer. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 87–107, 1962.

[88] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, Massachusetts, 2 edition, 1998.

[89] Lawrence Snyder. *A Programming Guide to ZPL*. MIT Press, Scientific and Engineering Computation Series, March 1999.

[90] Sun Microsystems, Inc., Burlington, Massachusetts. *The Fortress Language Specification, Version 0.707*, July 2005.

[91] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual, Version 1.0*, February 1991.

[92] Transputer: A Programmable Component that Gives Micros a New Name. *Computer Design*, 23:243–244, February 1984.

[93] R. Triolet, F. Irigoin, and P. Feautrier. Direct Parallelization of CALL Statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986.

[94] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed–Memory Machines*. PhD thesis, Rice University, January 1993.

[95] P. S. Tseng. A Systolic Array Programming Language. In *Fifth Distributed-Memory Computing Conference*, pages 1125–1130, Charleston, South Carolina, April 1990.

[96] M. Ujaldon, E.L. Zapata, B. Chapman, and H. Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, October 1997.

[97] Vax computers. `http://h18000.www1.hp.com/alphaserver/vax/timeline/1986.html`.

[98] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.

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

[100] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A Tool For Semi–Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.

[101] H. Zima and B. Chapman. Compiling for Distributed–Memory Systems. *Proceedings of the IEEE*, 81(2):264–287,

February 1993.

[102] Hans P. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification. *Internal Report 21, ICASE, NASA Langley Research Center*, March 1992.