A Library-Based Approach to Task Parallelism in a Data-Parallel Language

Ian Foster,*,[1] David R. Kohr, Jr.,*,[1] Rakesh Krishnaiyer,†,[2] and Alok Choudhary‡,[3]

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439; †Department of Computer and Information Science, Syracuse University, Syracuse, New York 13244; and ‡ECE Department, Technological Institute, Northwestern University, 2145 Sheridan Road, Evanston, Illinois 60208-3118

Pure data-parallel languages such as High Performance Fortran version 1 (HPF) do not allow efficient expression of mixed task/data-parallel computations or the coupling of separately compiled data-parallel modules. In this paper, we show how these common parallel program structures can be represented, with only minor extensions to the HPF model, by using a coordination library based on the Message Passing Interface (MPI). This library allows data-parallel tasks to exchange distributed data structures using calls to simple communication functions. We present microbenchmark results that characterize the performance of this library and that quantify the impact of optimizations that allow reuse of communication schedules in common situations. In addition, results from two-dimensional FFT, convolution, and multiblock programs demonstrate that the HPF/MPI library can provide performance superior to that of pure HPF. We conclude that this synergistic combination of two parallel programming standards represents a useful approach to task parallelism in a data-parallel framework, increasing the range of problems addressable in HPF without requiring complex compiler technology. © 1997 Academic Press

1. INTRODUCTION

The data-parallel language High Performance Fortran version 1 (abbreviated simply as HPF in the following) provides a portable, high-level notation for expressing data-parallel algorithms [17]. An HPF computation has a single-threaded control structure, global name space, and loosely synchronous parallel execution model. Many problems requiring high-performance implementations can be expressed succinctly in HPF.

However, HPF does not adequately address task parallelism or heterogeneous computing. Examples of applications that are not easily expressed using HPF alone [6, 14] include multidisciplinary applications where different modules represent distinct scientific disciplines, programs that interact with user

interface devices, applications involving irregularly structured data such as multiblock codes, and image-processing applications in which pipeline structures can be used to increase performance. Such applications must exploit task parallelism for efficient execution on multicomputers or on heterogeneous collections of parallel machines. Yet they may incorporate significant data-parallel substructures.

These observations have motivated proposals for the integration of task and data parallelism. Two principal approaches have been investigated. Compiler-based approaches seek to identify task-parallel structures automatically, within data-parallel specifications [11, 14, 21], while language-based approaches provide new language constructs for specifying task parallelism explicitly [3, 6, 19, 24]. Both approaches have shown promise in certain application areas, but each also has disadvantages. Compiler-based approaches complicate compiler development and performance tuning, and language-based approaches also introduce the need to standardize new language features.

In this paper, we propose an alternative approach to task/data-parallel integration, based on specialized coordination libraries designed to be called from data-parallel programs. These libraries support an execution model in which disjoint process groups (corresponding to data-parallel tasks) interact with each other by calling group-oriented communication functions. In keeping with the sequential reading normally associated with data-parallel programs, each task can be read as a sequential program that calls equivalent single-threaded coordination libraries. The potentially complex communication and synchronization operations required to transfer data among process groups are encapsulated within the coordination library implementations.

To illustrate and explore this approach, we have defined and implemented a library that allows the use of a subset of the Message Passing Interface (MPI) [13] to coordinate HPF tasks. MPI standardizes an interaction model that has been widely used and is well understood within the high-performance computing community. It defines functions for both point-to-point and collective communication among tasks executing in separate address spaces; its definition permits efficient imple-

---

[1]E-mail: {foster, kohr}@mcs.anl.gov
[2]E-mail: rakesh@cat.syr.edu
[3]E-mail: choudhar@ece.nwu.edu

mentations on both shared and distributed-memory computers [12]. Our HPF/MPI library allows these same functions to be used to communicate and synchronize among HPF tasks. This integration of two parallel programming standards allows us to incorporate useful new functionality into HPF programming environments without requiring complex new directives or compiler technology. We argue that the approach provides a conceptually economical and hence easily understood model for parallel program development and performance tuning.

In brief, the contributions of this paper are as follows:

1. The definition of a novel parallel programming model in which group-oriented communication libraries are used to coordinate the execution of process groups corresponding to data-parallel tasks.

2. The demonstration that an HPF binding for MPI allows the range of problems efficiently expressible in HPF to be extended without excessive conceptual or implementation complexity.

3. The illustration and evaluation using realistic applications of design techniques for achieving communication between data-parallel tasks, for integrating MPI library calls into HPF programs, and for exploiting information provided by MPI communication calls to improve communication performance.

A preliminary report on some of the techniques and results presented here appeared as [7]; the present paper provides a more detailed description of our techniques and introduces additional optimizations that improve performance by a factor of two or more in some situations.

The problem of parallel program coupling has been investigated by a number of other groups, although not in this standards-based fashion. Groups building multidisciplinary models frequently build specialized "couplers" responsible for transferring data from one model to another. Coupler toolkits have been proposed and built, but not widely adopted. MetaCHAOS [5] provides a more general coupling tool by defining a model in which programs can export and import distributed data structures; MetaCHAOS handles communication scheduling. These various efforts are complementary to the work reported here in that they could all benefit from the efficient communication mechanisms used in our HPF/MPI library, if the models in question were written in HPF.

In the rest of this paper, we describe the design and implementation of our HPF/MPI library, provide an example of its use, and evaluate its performance. In the implementation section, we focus on issues associated with point-to-point communication and describe techniques for determining data distribution information and for communicating distributed data structures efficiently from sender to receiver. We also show how specialized MPI communication functions can be used to trigger optimizations that improve performance in typical communication structures. We use microbenchmark experiments to quantify the costs associated with our techniques and the benefits of our optimizations. We also present results from multi-

block and two-dimensional fast Fourier transform (FFT) and convolution codes that demonstrate that HPF/MPI can indeed offer performance advantages relative to pure HPF.

2. DATA AND TASK PARALLELISM

We motivate our approach to the integration of task and data parallelism by discussing data parallelism and HPF and then reviewing approaches to the extension of the data-parallel model.

2.1. Data Parallelism and HPF

Data-parallel languages allow programmers to exploit the concurrency that derives from the application of the same operation to all or most elements of large data structures [15]. Data-parallel languages have significant advantages relative to the lower level mechanisms that might otherwise be used to develop parallel programs. Programs are deterministic and have a sequential reading. This simplifies development and allows reuse of existing program development methodologies—and, with some modification, tools. In addition, programmers need not specify how data are moved between processors. On the other hand, the high level of specification introduces significant challenges for compilers, which must be able to translate data-parallel specifications into efficient programs [1, 16, 22, 27].

High Performance Fortran [17] is perhaps the best-known data-parallel language. HPF exploits the data parallelism resulting from concurrent operations on arrays. These operations may be specified either explicitly by using parallel constructs (e.g., array expressions and FORALL) or implicitly by using traditional DO loops.

HPF addresses the problem of efficient implementation by providing directives that programmers can use to guide the parallelization process. In particular, distribution directives specify how data are to be mapped to processors. An HPF compiler normally generates a single-program, multiple-data (SPMD) parallel program by applying the *owner computes rule* to partition the operations performed by the program; the processor that "owns" a variable is responsible for updating its value [1, 22, 27]. The compiler also introduces communication operations when local computation requires remote data. An attractive feature of this implementation strategy is that the mapping from user program to executable code is fairly straightforward. Hence, programmers can understand how changes in program text affect performance.

We use a two-dimensional fast Fourier transform (2-D FFT) to illustrate the application of HPF. The HPF implementation presented in Fig. 1 calls the subroutine rowfft to apply a one-dimensional (1-D) FFT to each row of the 2-D array A, and then transposes the array and calls rowfft again to apply a 1-D FFT to each column. The 1-D FFTs performed within rowfft are independent of each other and can proceed in parallel. The PROCESSORS directive indicates that the program is to run on eight virtual processors; the DISTRIBUTE directive indicates that A is distributed

```
!HPF$ processors pr(8)
      complex A(8, 8)
!HPF$ distribute A(BLOCK,*)
      do i = 1, 100
            call read(A)
            call rowfft(8, A)
            A = transpose(A)
            call rowfft(8, A)
            call write(A)
      end do
```
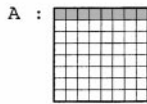
FIG. 1. An HPF implementation of a 2-D FFT, in this case configured to use eight processors and to operate on an array of size 8×8. Shading indicates the elements of the array A that are mapped to processor 0.

by row. This distribution allows the `rowfft` routine to proceed without communication. However, the transposition `A=transpose(A)` involves all-to-all communication.

2.2. Task Parallelism

Certain important program structures and application classes are not directly expressible in HPF [6, 14]. For example, both real-time monitoring and computational steering require that programmers connect a data-parallel simulation code to another sequential or parallel program that handles I/O. The simulation task periodically sends arrays to the I/O task, which processes them in some way (e.g., displays them) and perhaps also passes control information back to the simulation.

As a second example, we consider the 2-D FFT once again. Assume an array of size $N \times N$ and P processors. Because the computation associated with the FFT scales as $N^2 \log N$ while the communication due to the transpose scales only as max (N^2, P^2), the data-parallel algorithm described in Section 2.1 is efficient when N is much larger than P. However, signal-processing systems must often process quickly a stream of arrays of relatively small size. (The array size corresponds to the sensor resolution and might be 256×256 or less.) In these situations, an alternative pipelined algorithm is often more efficient [4, 14]. The alternative algorithm partitions the FFT computation among the processors such that $P/2$ processors perform the `read` and the first set of 1-D FFTs, while the other $P/2$ perform the second set of 1-D FFTs and the `write`. At each step, intermediate results are communicated from the first to the second set of processors. These intermediate results must be transposed on the way; since each processor set has size $P/2$, $P^2/4$ messages are required. In contrast, the data-parallel algorithm's all-to-all communication involves $P(P - 1)$ messages, communicated by P processors: roughly twice as many per processor.

These two examples show how both modularity and performance concerns can motivate us to structure programs as collections of data-parallel tasks. How are such task/data-parallel computations to be represented in a data-parallel language such as HPF? Two principal approaches have been proposed: implicit approaches based on compiler technology and explicit approaches based on language extensions or programming environments for task coordination.

Compiler-based approaches. Advocates of implicit, compiler-based approaches seek to develop more sophisticated compilers capable of extracting task-parallel algorithms from data-parallel specifications. Frequently, they will use new directives to trigger the application of specific transformations. This general approach has been used to exploit pipeline [14] and functional parallelism [21], for example. (A variant of the former approach has been incorporated in HPF version 2, but is not yet supported in commercial compilers.)

Implicit, compiler-based approaches maintain a deterministic, sequential reading for programs. However, these approaches also tend to increase the complexity of the mapping from user program to executable code. This increased complexity can be a disadvantage for both programmers and compiler writers. For programmers, it becomes more difficult to understand how changes in program source affect achieved performance, and hence more difficult to write efficient programs. For compiler writers, it becomes more difficult to build compilers that generate efficient code, particularly because optimization techniques for different constructs and situations tend to interact in complex ways.

Language-based approaches. Advocates of explicit, language-based approaches propose new language constructs that allow programmers to specify the creation and coordination of tasks explicitly. The basic concept is that of a coordination language [2, 9], except that because the tasks are themselves data-parallel programs, we obtain a hierarchical execution model in which task-parallel computation structures orchestrate the execution of multiple data-parallel tasks.

Language-based approaches have been proposed that use a graphical notation [3], channels [6], remote procedure calls [19], and a simple pipeline notation [24] to connect data-parallel computations. Promising results have been obtained. Nevertheless, there is as yet no consensus on which language constructs are best. Since successful adoption depends on consensus and then standardization, language-based approaches clearly are not a near-term solution.

3. AN HPF BINDING FOR MPI

Explicit task-parallel coordination libraries represent an alternative approach to the integration of task and data parallelism that avoids the difficulties associated with compiler-based and language-based techniques. We use the example of an HPF binding for MPI to illustrate the approach and to explore practical issues associated with its implementation.

MPI provides a set of functions, data types, and protocols for exchanging data among and otherwise coordinating the execution of multiple tasks; a "binding" defines the syntax used for MPI functions and data types in a particular language. Previous MPI implementations have supported bindings only for the sequential languages C and Fortran 77 [12]. However, there is no reason why MPI functions may not also be used for communication among data-parallel tasks. Our HPF binding

for MPI makes this possible. It is intended to be used as follows:

• A programmer initiating a computation requests (using some implementation-dependent mechanism) that a certain number of tasks be created; each task executes a specified HPF program on a specified number of processors.

• Tasks can call MPI functions to exchange data with other tasks, using either point-to-point or collective communication operations. In point-to-point communications, a sender and a receiver cooperate to transfer data from sender to receiver; in collective communications, multiple tasks cooperate—for example, to perform a reduction.

When reading HPF/MPI programs, HPF directives can be ignored, and code understood as if it implements a set of sequential tasks that communicate using MPI functions. The source and destination arguments that appear in MPI calls denote IDs of the corresponding tasks involved in HPF/MPI.

Figure 2 uses HPF/MPI to implement the pipelined 2-D FFT algorithm described in Section 2.2. Task 0 calls `rowfft` to apply a 1-D FFT to each row of the array A (8×8 complex numbers, distributed by row) and then calls the MPI function `MPI_Send` to send the contents of A to task 1. Task 1 implements the transpose by using `MPI_Recv` to receive this data from task 0 into an array B, distributed by column, and then calls a subroutine `colfft` to apply a 1-D FFT to each column. The value 99 is a message tag.

A comparison with Fig. 1 shows that the HPF/MPI version is not significantly more complex. In essence, we have replaced the transpose in the HPF program with two subroutine calls. Notice that these calls specify only the logical transfer of data from one data-parallel task to another: the potentially complex communication operations required to achieve this transfer are encapsulated within the HPF/MPI library. This example
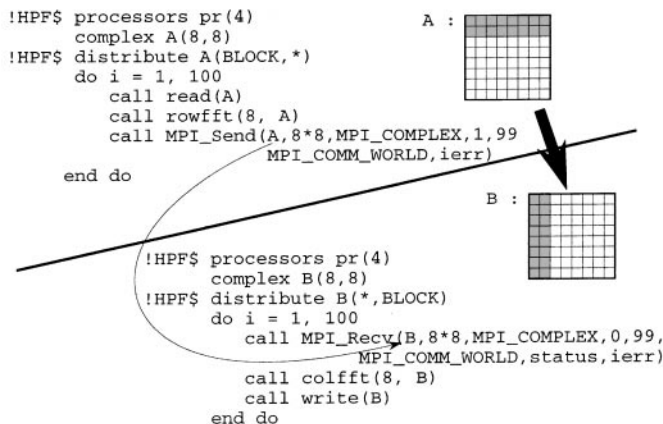
illustrates how a coordination library can gain leverage from a data parallel language's high-level support for the management of distributed data structures and associated index translation operations, while providing an explicit, easily understood notation for specifying task-parallel computations. In more complex situations—such as multiblock codes—an HPF/MPI formulation can actually be more succinct than a pure HPF version.

4. IMPLEMENTATION

A number of factors influenced the design of our prototype implementation of HPF/MPI. For example, we wanted our library to be portable among different hardware platforms, and to be able to operate with different HPF compilation systems. At the same time, we wanted typical HPF/MPI applications to achieve good performance with only modest effort by the programmer.

4.1. Design Overview

We now describe the techniques that we have developed to address these requirements. For brevity, we examine only the case of point-to-point operations on distributed-memory multicomputers; elsewhere we discuss techniques for implementing other operations [8]. Figure 3 illustrates the basic processing steps performed by our library for a single point-to-point transfer. The actions taken by senders and receivers are symmetrical, so it suffices to examine just the processing steps of a send operation. These seven steps are as follows:

1. *Distribution inquiry*. Standard HPF inquiry intrinsics such as `HPF_DISTRIBUTION` are called to determine the distribution of the array being sent.

2. *Extrinsic call*. The portion of the library that is written in HPF calls a coordination library function that is written in C and declared as *extrinsic* (foreign) to HPF. This causes the execution model of each processor in the task to change from data-parallel (globally single-threaded) to SPMD (separate
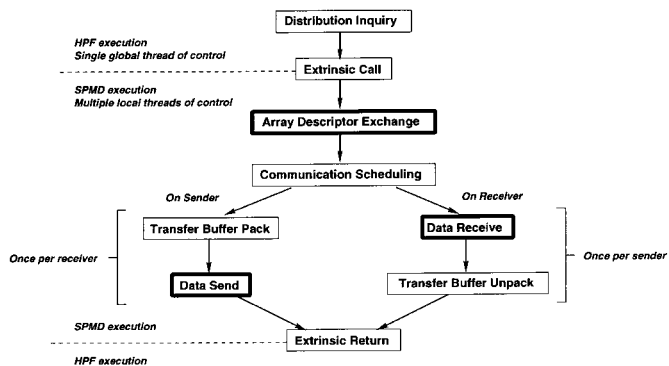

FIG. 2. HPF/MPI implementation of a task/data-parallel pipelined 2-D FFT configured as two tasks, each on four processors and operating on arrays of size 8×8. Shading indicates array elements mapped to processor 0 in task 0 and in task 1. Note that the arrays A and B are mapped to disjoint sets of processors.

FIG. 3. The steps executed during an HPF/MPI point-to-point transfer. The thick boxes distinguish the steps that require communication. The sending and receiving sides differ only in the fifth and sixth steps.

threads of control on each processor, as in HPF's *local* mode of execution [17]).

3. *Array descriptor exchange.* Sending processors exchange distribution information with receiving processors about the source and destination arrays. After Step 1, all senders have distribution descriptors for the source array and all receivers have descriptors for the destination. We exploit this fact to avoid expensive broadcast operations and instead perform pairwise exchanges between individual senders and receivers.

4. *Communication scheduling.* Sending processors use the distribution information obtained in Step 3 to compute *communication schedules*, that is, the subsections of the source array that should be sent to each receiving processor.

5. *Transfer buffer pack.* Using the communication schedule computed in Step 4, we pack the array elements required by a particular receiver into a contiguous communication buffer.

6. *Data send.* The contents of the buffer packed in Step 5 are sent to the corresponding receiver.

7. *Extrinsic return.* By returning from the extrinsic function called in Step 2, the execution model of each processor reverts to data-parallel, so that execution of the HPF program may resume.

Steps 5 and 6 are repeated once for each processor to which data must be sent. The order in which each sender transfers array subsections to each receiver is chosen so as to maximize parallelism among the individual transfers; a detailed description of this ordering appears in [18].

4.2. Implementation Details

Based on the above design, we have implemented a prototype HPF/MPI library that supports a subset of MPI's point-to-point communication functions. This prototype operates with the commercial HPF compiler `pghpf` (version 2.0), developed by the Portland Group, Inc. [25]. Because of our desire for portability, we defined a run-time initialization interface between `pghpf` and HPF/MPI that minimizes the dependence of HPF/MPI upon the internals of the HPF runtime system. The interface establishes separate MPI communicators for each HPF task and for HPF/MPI, so that the communications of the HPF tasks and HPF/MPI cannot interfere with one another. We believe that this interface will work also with other HPF compilation systems that use MPI for communications.

In some circumstances, it is desirable to reduce the total volume of communicated data by sending only a portion of an array, rather than an entire array. HPF permits programmers to denote portions of arrays using array section notation. Our implementation of HPF/MPI accepts array sections as the source or destination of a point-to-point operation. As an example, the following call sends just the first row of the source array A:

```
call MPI_Send(A(1, :), N, MPI_FLOAT,
              1, 99, MPI_COMM_WORLD)
```

While developing HPF/MPI, we encountered design choices in which one must make tradeoffs between portability and performance. The tradeoffs center around whether HPF/MPI accesses distributed arrays using the portable extrinsic call mechanism, which copies arrays between the nonportable layout of a particular HPF compiler and the portable, contiguous layout used by C and Fortran 77. A system that does not use extrinsic calls, and instead accesses arrays directly in HPF's internal representation, saves data copying at the cost of portability. We have implemented two different versions of HPF/MPI, one called "non-DIRECT" which uses extrinsic calls, and another ("DIRECT") which avoids extrinsic calls by directly accessing arrays. In the next section we quantify the overhead of using the extrinsic call mechanism.

Communication schedules are generated in Step 4 using algorithms based on the FALLS (FAmiLy of Line Segments) distributed array representation of Ramaswamy and Banerjee [20]. These algorithms compute the minimal sets of array elements that must be transferred from sending to receiving processors. The algorithms rely on modulo arithmetic and are highly efficient: for typical redistributions, their running time is proportional to the number of participating processors. As we shall see in the next section, schedule computation never constitutes more than a small fraction of total transfer time.

MPI provides programmers with facilities for optimizing communication between processors. Many of these facilities are useful in the context of intertask communication also. For example, the functions `MPI_Send_init` and `MPI_Recv_init` define what are called *persistent requests* for point-to-point operations; once defined, a request can be executed repeatedly using `MPI_Start`. As illustrated in Fig. 4, MPI programmers can use these functions to indicate that the same data transfer will be performed many times. Our HPF/MPI implementation of these calls computes a communication schedule just once, when the request is defined. Subsequent calls to `MPI_Start` reuse the schedule, so that costs associated with Steps 1, 3, and 4 can be amortized over many transfers. In [8] we discuss how other MPI optimization features could be incorporated into HPF/MPI.

```
!HPF$ processors pr(4)
      complex A(8,8)
      integer request
!HPF$ distribute A(BLOCK,*)
      call MPI_Send_init(A,8*8,MPI_COMPLEX,1,99,
                          MPI_COMM_WORLD,request,ierr)
      do i = 1, 100
         call read(A)
         call rowfft(8, A)
         call MPI_Start(request,ierr)
      end do
```

FIG. 4. An alternative HPF/MPI formulation of the sending side of the pipelined 2-D FFT, in which `MPI_Send_init` is used to define a persistent request that is then executed repeatedly by `MPI_Start`.

```
!HPF$ processors pr(P)
      real From(N,N), To(N,N)
!HPF$ distribute From(BLOCK,*), To(*,BLOCK)
      call MPI_Init(ierr)
      call MPI_Comm_Rank(MPI_COMM_WORLD,myid,ierr)
      if (myid .eq. 0) then
         do i = 1, 100
            call MPI_Send(From,N*N,MPI_REAL,1,99,
                          MPI_COMM_WORLD,ierr)
            call MPI_Recv(To,N*N,MPI_REAL,1,99,
                          MPI_COMM_WORLD,status,ierr)
         end do
      else
         do i = 1, 100
            call MPI_Recv(To,N*N,MPI_REAL,0,99,
                          MPI_COMM_WORLD,status,ierr)
            call MPI_Send(From,N*N,MPI_REAL,0,99,
                          MPI_COMM_WORLD,ierr)
         end do
      endif
      call MPI_Finalize(ierr)
      end
```

FIG. 5. The microbenchmark used to quantify HPF/MPI communication costs. This program is intended to execute as two tasks. `MPI_Init` and `MPI_Finalize` set up and shut down the MPI library, respectively, while `MPI_Comm_rank` returns the rank of the calling task (0 or 1 in this case).

5. PERFORMANCE STUDIES

We use a simple microbenchmark to quantify the costs associated with the implementation scheme just described. This "ping-pong" program, presented in Fig. 5, repeatedly exchanges a 2-D array of fixed size between two tasks. The array is distributed (`BLOCK,*`) in the sender and (`*,BLOCK`) in the receiver, which induces a worst-case communication pattern in which all senders must communicate with all receivers.

We run the benchmark using tasks of varying size exchanging both small (4 KB) and large (4 MB) arrays. This allows us to determine how the cost components of transfer operations vary with task and array size. We measure three different versions of the benchmark: one that uses neither persistent operations nor direct access to HPF arrays ("Nonpersistent/Nondirect"), one that uses persistent operations but not direct access ("Persistent/Nondirect"), and one that uses both persistent operations and direct access ("Persistent/Direct"). By comparing these different versions, we can gauge the effectiveness of the persistent operation optimization and the cost of the extrinsic call mechanism.

All experiments are performed on the Argonne IBM SP2, which contains 128 Power 1 processors connected by an SP2 multistage crossbar switch. We record the maximum execution time across all processors. As the underlying sequential communication library we use the portable MPICH implementation of MPI.

5.1. Description of Results

The plots of Fig. 6 show the resulting measurements. Each vertical bar represents the one-way transfer time obtained from one experiment, and the shaded regions within each bar represent the fraction of time spent in the processing steps described in the previous section. For brevity, we have combined into one shaded region the times for corresponding steps in the sender and receiver. In addition, pack and unpack are combined as Message Assembly, and send and receive are labeled Data Transfer. We have also merged Extrinsic Return into Extrinsic Call.

In studying these results, we first note that for small problem sizes (N), the total cost increases with the number of processors (P), while for large N, total time decreases with P. These results are to be expected: for small N, the dominant contributor to total communication cost is the message startup time, or latency, which increases with P; for large N, the dominant contributor is the message transfer time, which is proportional to message length and therefore decreases with P.

5.2. Processing Step Costs

We now analyze the costs related to each of the processing steps. Steps 1, 3, and 4 are associated with determining how to perform a communication, and their costs are amortized over repeated transfers if persistent communications are used. These three cost components are shown uppermost in each bar, which in most cases allows us to distinguish the costs for nonpersistent and persistent communication. By comparing the Nonpersistent/Nondirect cases with the Persistent/Nondirect cases, we see that for small messages, using persistent operations results in a savings of up to 40% of the total time. The savings for large messages is negligible, because per-byte transfer costs dominate the total time.

We note that the time for Step 3 (Array Descriptor Exchange) includes synchronization delays resulting from extra processing performed at receiving processors in other steps, such as communication and buffer unpacking at the end of the receive. Hence the high Step 3 times for large N and small P in the Nonpersistent/Nondirect case are an artifact of the experimental protocol, not a sign of inefficiency in the implementation of descriptor exchange. A similar synchronization effect causes increased times for Data Transfer in the two persistent cases.

Step 2 (Extrinsic Call) represents the costs associated with the extrinsic call mechanism. This component represents a fixed cost for multiple subroutine calls, plus a per-byte overhead for copying array data between HPF's memory layout and a contiguous layout. For $P = 1$ and an array of size 4 KB, Step 2 costs about 350 μs; for $P = 1$ and a 4 MB array, the cost is about 36 ms. These data suggest a fixed cost of roughly 300 μs and an incremental cost of about 0.0086 μs/byte (116 MB/s copy bandwidth). Because the source array in the ping-pong benchmark is an input argument to the send operation, and is not changed between sends, `pghpf` optimizes the extrinsic call by performing a copy during the extrinsic call of just the first send operation. In contrast, a copy must be performed during the extrinsic return step of each receive operation. Therefore the per-byte costs of Extrinsic Call in Fig. 6 reflect copying only on the receiving side.

By comparing the Persistent/Nondirect and Persistent/Direct cases, we can evaluate the benefit of avoiding the extrinsic
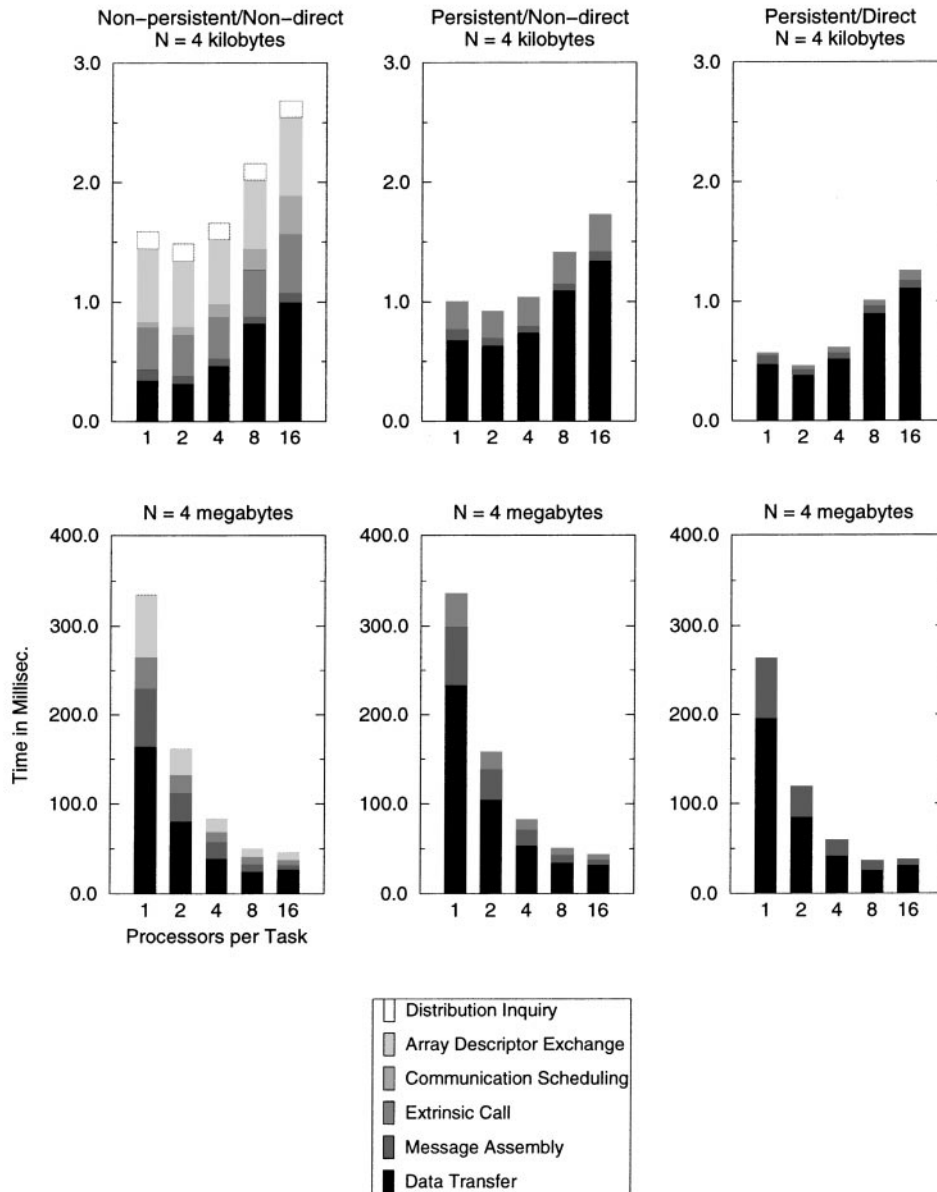
FIG. 6. Time required for a one-way HPF/MPI point-to-point communication on an IBM SP2, for various array sizes, task sizes, and implementation versions.
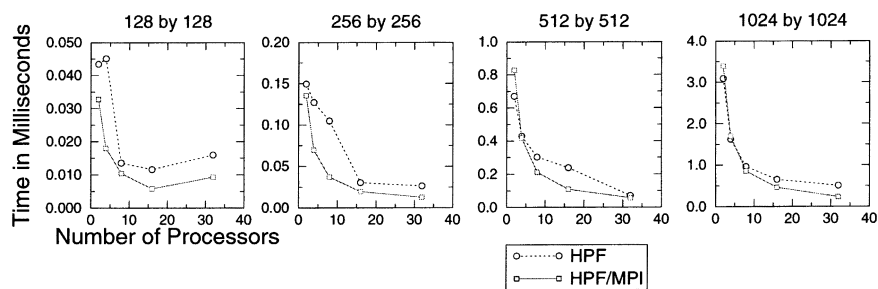

FIG. 7. Execution time per input array for HPF and HPF/MPI implementations of the 2-D FFT application, as a function of the number of processors. Results are given for different problem sizes.
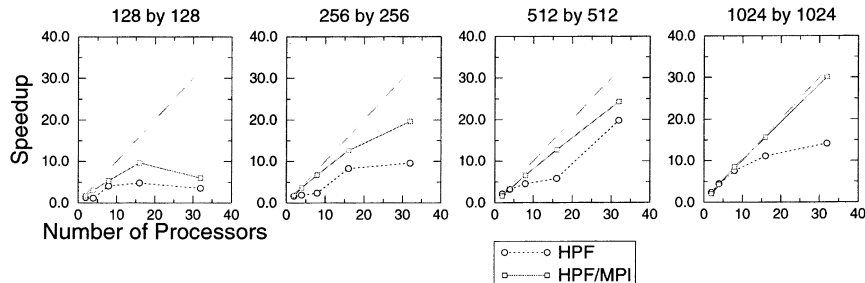
FIG. 8. Speedup obtained for HPF and HPF/MPI implementations of the 2-D FFT application, as a function of the number of processors.

call mechanism. For small arrays, elimination of the fixed extrinsic call costs improves performance by up to 30%. For large arrays, elimination of the copying performed during an extrinsic call provides improvements of up to 20%.

Step 5 (buffer pack/unpack) corresponds to the costs of assembling messages from potentially noncontiguous locations before transmission, and disassembling them upon reception. Our implementation performs this assembly and disassembly explicitly in all cases; optimized implementations might be able to avoid this extra copying for some distributions on some platforms. For large messages the pack/unpack steps execute at a rate of about 64 MB/s. As we would expect, this is about half the rate achieved for the Extrinsic Call step, which performs copying in the receiver but not the sender.

The final cost component is the actual communication (the Data Transfer shaded region). Since our transfer strategy permits senders to perform their transfers to receivers in parallel, we expect that the execution time of intertask transfers is governed by $Pt_s + (N/P)t_b$, where t_s is the per-message startup cost, N is the amount of data in the array (in bytes), and t_b is the per-byte data transfer time. The experimental data fit this simple model reasonably well. A more detailed model and more extensive analysis appear in [18].

5.3. Performance Summary

For large arrays, HPF/MPI achieves a bandwidth of about 12 MB/s in the two nondirect cases, and up to about 17 MB/s in the Persistent/Direct case. The underlying MPICH library can transfer data at a maximum rate of about 30 MB/s on the SP. Hence HPF/MPI achieves roughly half the bandwidth available on this platform. The data transfer rate for large arrays during the Data Transfer step is about 25 MB/s per sender–receiver processor pair, which indicates that transfers are proceeding in parallel at close to the maximum rate. The degradation in overall bandwidth in HPF/MPI compared to MPICH is due chiefly to the extra copying in the extrinsic call and buffer pack/unpack steps.

In summary, the microbenchmark results show that the persistent communication optimization provides significant benefits when transferring small arrays; that our HPF/MPI implementation achieves reasonable performance for small arrays when the persistent communication optimization is applied, and for large arrays in all cases; and that a considerable per-

formance improvement is realized by directly manipulating arrays stored in HPF's internal representation.

6. APPLICATIONS

We also studied the performance of HPF/MPI implementations of application kernel benchmarks like 2-D FFT, 2-D convolution, and multiblock codes, comparing each with an equivalent pure HPF program. In each case, we employ the persistent communication optimization when transferring data between tasks. Our results demonstrate that in most instances the HPF/MPI library achieves performance superior to that of pure HPF.

6.1. 2-D FFT

The HPF/MPI and HPF implementations are based on the codes given in Figs. 2 and 1, respectively. For our experiments, we replace the `read` call in the 2-D FFT with a statement that initializes array `A`, and eliminate the `write` call entirely. The code was tuned for good cache performance with an experimentally-determined blocking parameter. The HPF/MPI code is executed as a pipeline of two tasks, with an equal number of processors assigned to each task. Figure 7 presents our results, which are performed for a number of images large enough to render pipeline startup and shutdown costs insignificant. The execution times shown are the average per image. The speedup obtained over a sequential version of the code is shown in Fig. 8. The performance of the HPF/MPI version is generally better. In particular, for a fixed image size, HPF/MPI provides an increasing improvement in speedup as P increases.
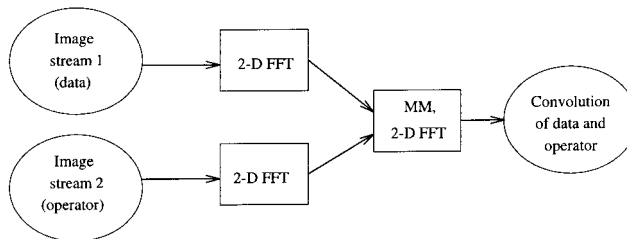

FIG. 9. Convolution algorithm structure. Two image streams are passed through forward FFTs and then to a pointwise matrix multiplication (MM) and inverse FFT.
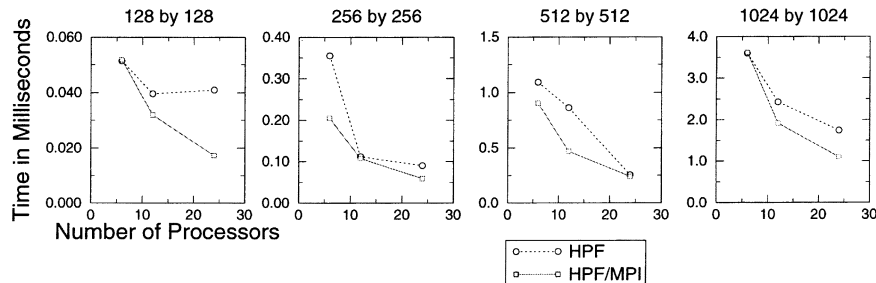
FIG. 10. Execution time per input array for HPF and HPF/MPI implementations of convolution, as a function of the number of processors. Results are given for different problem sizes.

6.2. 2-D Convolution

Convolution is a standard technique used to extract feature information from images [4, 23]. Images, represented as arrays of size $N \times N$, are input in pairs on two streams, and convolution generates a single output stream of images of the same size. A single convolution operation involves transformation of the two input arrays using independent 2-D FFTs, a pointwise multiplication of the two transformed arrays, and the application of an inverse 2-D FFT on the resulting array to generate an output image (Fig. 9). A data-parallel convolution algorithm performs these steps in sequence for each pair of input images, while a pipelined algorithm can execute each rectangular block in Fig. 9 as a separate module. As in the 2-D FFT, this pipeline structure can improve performance by reducing the number of messages. Moreover, each module involves two 1-D FFTs, which are further pipelined as explained in the previous section.

The HPF/MPI code consists of six tasks (a column task and a row task for each of the three modules), each of size $P/6$, where P is the total number of processors available for each experiment. The values of P were chosen to provide 1, 2, or 4 processors per task for the HPF/MPI version. Figure 10 shows our results. The graph compares the average of the total elapsed time between HPF and HPF/MPI for performing 2-D convolution on one data set. Once again, we see that the HPF/MPI version is often significantly faster than the pure HPF version. On the largest image size plotted (1024×1024), HPF/MPI provides an improvement of up to 37% over pure HPF. A comparison of the speedups is shown in Fig. 11.

6.3. Multiblock

Multiblock codes decompose a complex geometry into multiple simpler blocks [26]. A solver is run within each block, and boundary data are exchanged between blocks periodically. For our experiments, we use a program that applies a simple Poisson solver within each block and that supports only simple geometries [10]. For ease in HPF implementation, we fixed the number of blocks to 3. We chose a geometry such that each block is square, but the middle block has one-fourth the area of the end blocks. For example, the largest geometry in our experiment has end blocks of size 512×512 and a middle block of size 256×256. We chose values of P such that fewer processors were assigned the smaller middle block under HPF/MPI. In particular, for $P = 5$, two processors work on the end blocks and one on the middle (a mapping of 2/1/2); for $P = 9$ the mapping is 4/1/4; and for $P = 18$ the mapping is 8/1/8.

We compare the performance of an HPF program that computes each of the three blocks in turn and an HPF/MPI program in which three tasks compute the three blocks concurrently. In the HPF version, each block is represented as one array which is distributed over all the available processors. In the HPF/MPI code, each task executes one block, and processors are allocated to blocks in proportion to their size. The blocks were distributed in a (`*, BLOCK`) fashion for both HPF and HPF/MPI codes. Figures 12 and 13 show our results. The HPF/MPI program is always faster than the pure HPF program. This application is more communication intensive than the other two applications. The superior performance of the HPF/MPI code is due to lower communication overhead and better scalability.
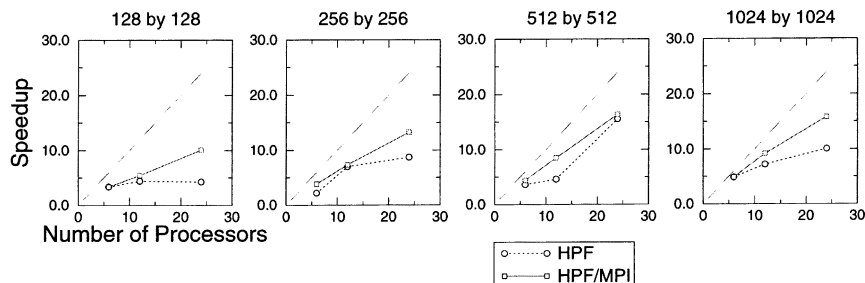

FIG. 11. Speedup obtained for HPF and HPF/MPI implementations of convolution, as a function of the number of processors.
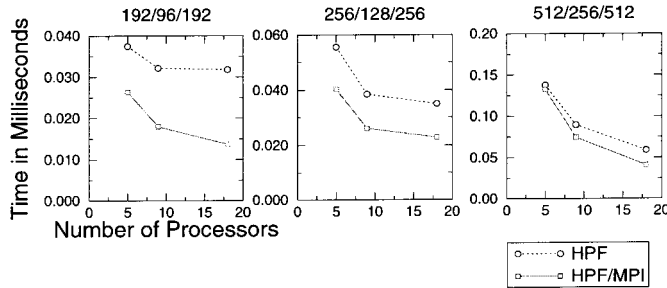
FIG. 12. Execution time for HPF and HPF/MPI implementations of the multiblock code, as a function of the number of processors.

7. CONCLUSIONS

An HPF binding for MPI can be used to construct task-parallel HPF applications and to couple separately compiled data-parallel programs, without a need for new compiler technology or language extensions. Our implementation of this binding executes efficiently on multicomputers, allowing us to write task/data-parallel 2-D FFT, convolution, and multiblock codes that execute faster than equivalent codes developed in HPF alone. On the basis of these results, we argue that the combination of the HPF and MPI standards provides a useful and economical approach to the implementation of task/data-parallel computations.

Microbenchmark results reveal various overheads associated with the HPF/MPI library. The MPI persistent request facility can be used to trigger optimizations that avoid overheads associated with exchange of distribution information and the computation of communication schedules. Overheads associated with the HPF extrinsic interface can be avoided by providing direct access to the internal representation used for HPF arrays. It is a topic for future research to determine the extent to which performance can be improved further by a tighter coupling between HPF/MPI and `pghpf`, by refining the HPF extrinsic interface, and by using compiler-derived information to select specialized communication functions.

The ideas developed in this paper can be extended in a number of ways. It appears likely that similar techniques can be used to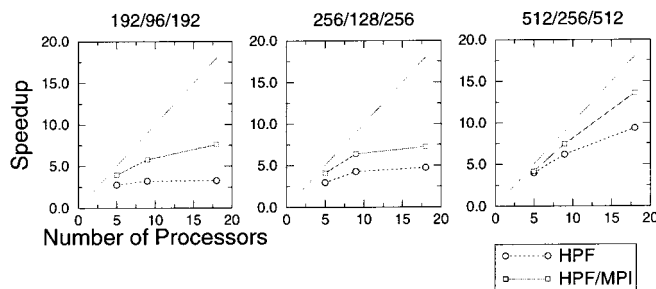 support other task interaction mechanisms. MPI and HPF extensions also suggest directions for further work. For example, MPI extensions proposed by the MPI Forum support client-server structures, dynamic task management, and single-sided operations. These constructs could be incorporated into an HPF/MPI system to support, for example, attachment to I/O servers and asynchronous coupling. Similarly, proposed support for mapping constructs within HPF (task regions) would allow the creation of task-parallel structures within a single program, by using HPF/MPI calls to communicate between task regions.

REFERENCES

1. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *J. Supercomputing* **2** (Oct. 1988), 151–169.

2. N. Carriero and D. Gelernter. Linda in context. *Commun. ACM* **32**(4) (Apr. 1989), 444–458.

3. G. Cheng, G. Fox, and K. Mills. Integrating multiple programming paradigms on Connection Machine CM5 in a dataflow-based software environment. Technical report, Northeast Parallel Architectures Center, Syracuse University, 1993.

4. A. N. Choudhary, B. Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for pipeline computations. *IEEE Trans. Parallel Distrib. Systems* **5**(4) (1994), 439–445.

5. G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries with Meta–Chaos. Technical Reports CS-TR-3633 and UMIACS-TR-96-30, Department of Computer Science, University of Maryland, and UMIACS, May 1996.

6. I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M. *Proceedings of the 1994 Scalable High Performance Computing Conference.* IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 293–300.

7. I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. *Proceedings of Supercomputing '96.* ACM Press, New York, 1996.

8. I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. MPI as a coordination layer for communicating HPF tasks. *Proceedings of the 1996 MPI Developers Conference.* IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 68–78.

9. I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

10. K. S. Gatlin and S. B. Baden. Brick: A benchmark for irregular block structured applications. Technical report, Department of Computer Science and Engineering, University of California at San Diego, 1996.

11. M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Systems* **3**(2) (1992), 166–178.

12. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard.

FIG. 13. Speedup obtained for HPF and HPF/MPI implementations of the multiblock code, as a function of the number of processors.

Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1996.

13. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Processing with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

14. T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel Distrib. Technol.* **2**(2) (Fall 1994), 16–26.

15. W. Hillis and G. Steele, Jr. Data parallel algorithms. *Commun. ACM* **29**(12) (1986), 1170–1183.

16. S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM* **35**(8) (Aug. 1992), 66–80.

17. C. Koelbel, D. Loveman, R. Schreiber, G. Stelle, Jr., and M. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, 1994.

18. D. R. Kohr, Jr. Design and optimization of coordination mechanisms for data-parallel tasks. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1996. [Online at http://www.mcs.anl.gov/fortran-m.]

19. P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. ICASE Report 94-39, Institute for Computer Application in Science and Engineering, Hampton, VA, May 1994.

20. S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation.* McLean, VA, February 1995, pp. 342–349.

21. S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting data and functional parallelism on distributed memory multicomputers, Technical Report CRHC-94-10, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1994.

22. A. Rogers and K. Pingali. Process decomposition through locality of reference. *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation.* Portland, OR, June 1989. ACM Press, New York.

23. A. Rosenfeld and A. Kak. *Digital Picture Processing.* Academic Press. New York, 1976.

24. B. Seevers, M. Quinn, and P. Hatcher. A parallel programming environment supporting data-parallel modules. *Internat. J. Parallel Programming* **21**(5) (Oct. 1992).

25. The Portland Group, Inc. *pghpf Reference Manual.* Wilsonville, OR.

26. V. N. Vatsa, M. D. Sanetrik, and E. B. Parlette. Development of a flexible and efficient multigrid-based multiblock flow solver; AIAA-93-0677. *Proc. 31st Aerospace Sciences Meeting and Exhibit,* Jan. 1993.

27. H. Zima, H.-J. Bast, and M. Gerndtd. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Comput.* **6** (1988), 1–18.

IAN FOSTER received his Ph.D. in computer science from Imperial College in 1988. He is currently a scientist in the Mathematics and Computer Science Division of Argonne National Laboratory, and associate professor of Computer Science at the University of Chicago. His research interests include languages, software tools, and applications of parallel computers, and the techniques required to integrate high-performance computers into networked environments. He recently served as software architect for the I-WAY distributed computing experiment.

DAVID R. KOHR, JR. graduated in 1988 from Washington University in St. Louis with a B.S. in computer science. From 1988 to 1991 he was on the staff of the MIT Lincoln Laboratory, developing software for radar data acquisition and signal processing. From 1991 to 1994 he was a graduate student at the University of Illinois at Urbana-Champaign, where he investigated tools and techniques for performance analysis of parallel application and system software, and from which he earned an M.S. in computer science. Since 1994 Kohr has been with Argonne National Laboratory, where he performs research on parallel run-time library support for communication, multithreading, and input–output.

RAKESH KRISHNAIYER is a Ph.D. candidate in computer science at Syracuse University. He received his M.S. in the same field from Syracuse University in 1996 and his B.Tech in computer science and engineering from the Indian Institute of Technology, Madras in 1993. He is currently pursuing his research in the Mathematics and Computer Science Division of Argonne National Laboratory. His research interests are in parallel and distributed computing, compilers, languages, and networks. He is a member of the IEEE Computer Society and the ACM.

ALOK CHOUDHARY received his Ph.D. from University of Illinois, Urbana-Champaign, in electrical and computer engineering, in 1989, and his M.S. from University of Massachusetts, Amherst, in 1986. He has been an associate professor in the Electrical and Computer Engineering Department at Northwestern University since September 1996. Alok Choudhary received the National Science Foundation's Young Investigator Award in 1993 (1993–1999). His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing, and scientific computing. Alok Choudhary served as the conference co-chair for the International Conference on Parallel Processing, and is currently the chair of the International Workshop on I/O Systems in Parallel and Distributed Systems.