Double Standards: Bringing Task Parallelism to HPF via the Message Passing Interface

Preprint ANL/MCS-P593-0596

Ian Foster† David R. Kohr, Jr.† Rakesh Krishnaiyer‡ Alok Choudhary§

†Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
U.S.A.
{foster,kohr}@mcs.anl.gov

‡Department of Computer and Information Science
Syracuse University
Syracuse, NY 13244
U.S.A.
rakesh@cat.syr.edu

§Department of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244
U.S.A.
choudhar@cat.syr.edu

Abstract

High Performance Fortran (HPF) does 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 a coordination library implementing the Message Passing Interface (MPI) can be used to represent these common parallel program structures. 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.

1 Introduction

High Performance Fortran (HPF) provides a portable, high-level notation for expressing data-parallel algorithms [14]. 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 [5, 11] 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 [8, 11, 16], while language-based approaches provide new language constructs for specifying task parallelism explicitly [3, 5, 15, 20]. Both approaches have shown promise in certain application areas, but each also has disadvantages. Compiler-based approaches complicate compiler development and performance tuning, while 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) [10] 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 implementations on both shared and distributed-memory computers [9]. 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 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 multiblock and two-dimensional fast Fourier transform (FFT) and convolution codes that demonstrate that HPF/MPI can indeed offer performance advantages relative to pure HPF.

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.

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 [12]. 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 is 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, 13, 18, 22].

High Performance Fortran [14] 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 is 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 value is responsible for updating its value [1, 18, 22]. The compiler also introduces communication operations when local computation requires remote data. An attractive feature of this implementation strategy

```
!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
```

A :

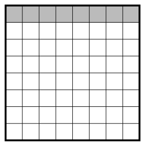

Figure 1: An HPF implementation of a 2-D FFT, in this case configured to use 8 processors and to operate on an array of size 8×8. Shading indicates the elements of the array `A` that are mapped to processor 0.

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 Figure 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 8 virtual processors; the `DISTRIBUTE` directive indicates that `A` is distributed 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 [5, 11]. 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. One example of an application domain in which such dynamic control is desirable is automotive design. Figure 2 depicts static output from an HPF implementation of the CHAD code, in which the air velocity tracers (arrows) were generated in a time-consuming postprocessing phase. We plan to use our HPF/MPI library to introduce a computational steering capability that allows scientists to place and visualize tracers dynamically, during program execution.

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,

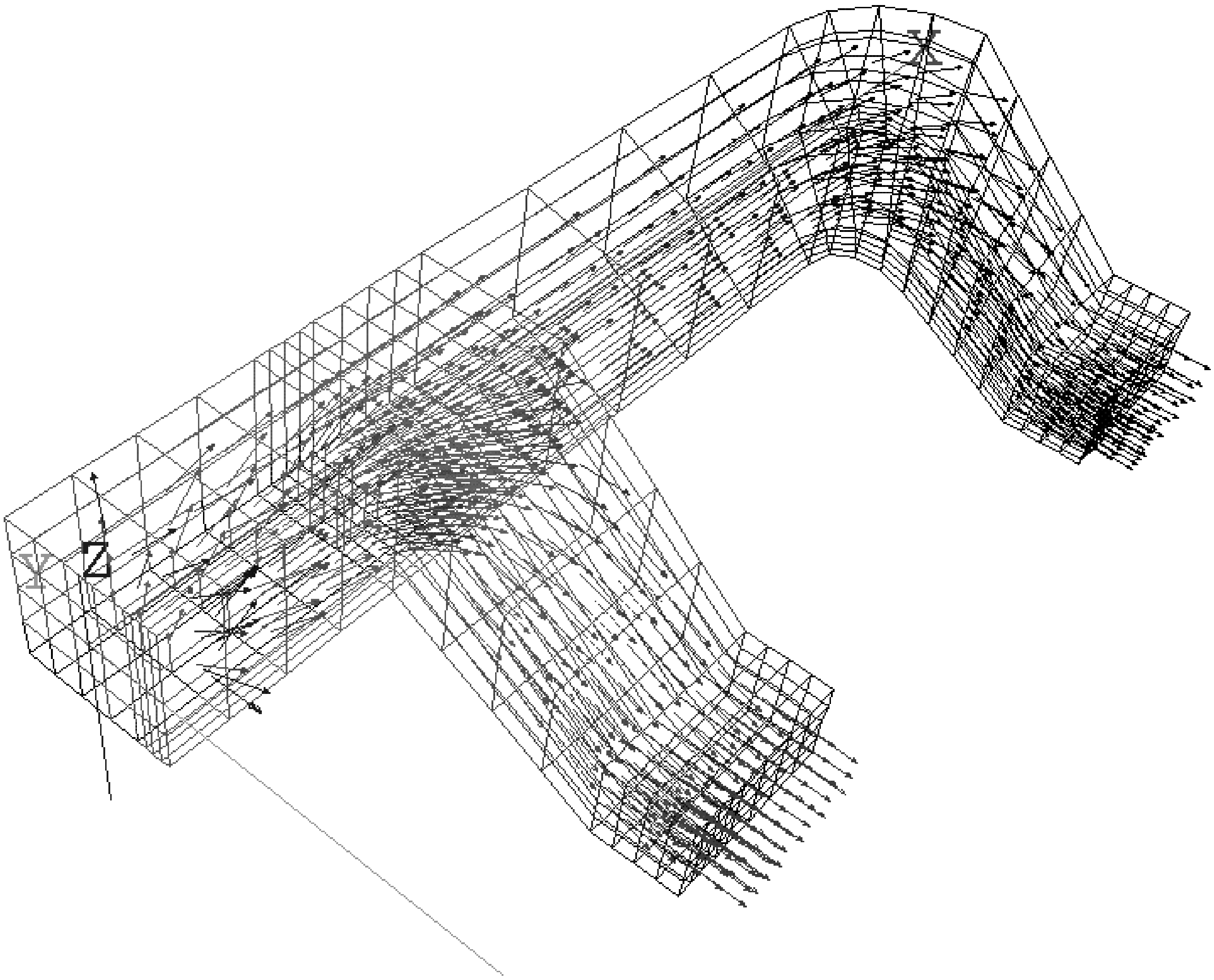

Figure 2: Air velocities in a passenger vehicle duct, as computed by the CHAD fluid dynamics program (image courtesy of T. Canfield)

an alternative pipelined algorithm is often more efficient [4, 11]. 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 [11] and functional parallelism [16], for example.

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, 6], 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 [5], remote procedure calls [15], and a simple pipeline notation [20] 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.

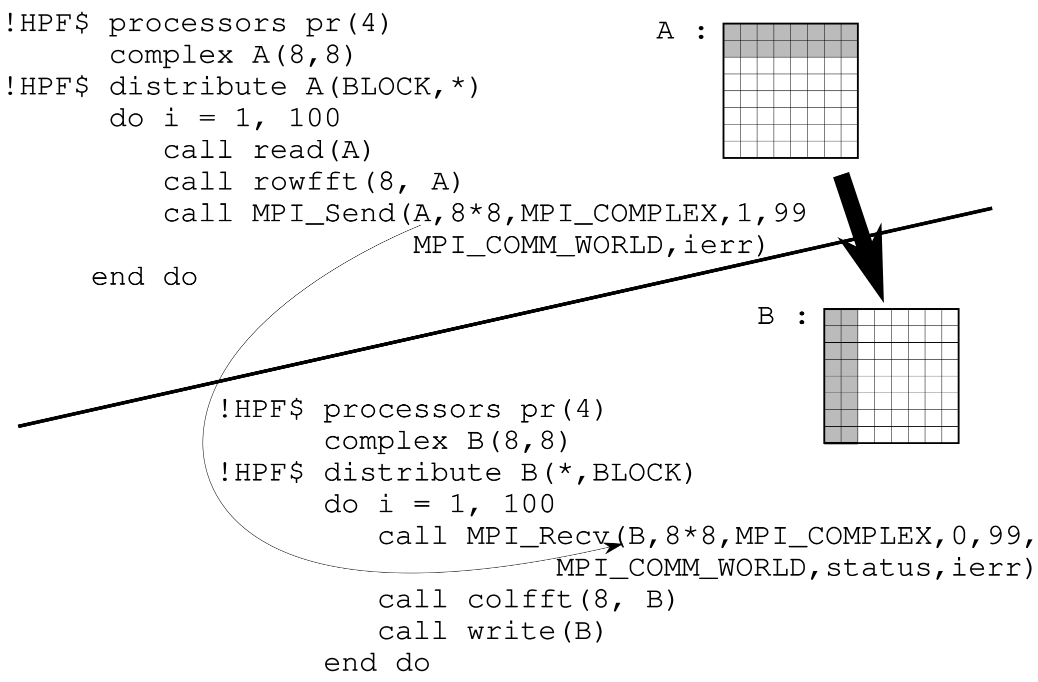

Figure 3: HPF/MPI implementation of a task/data-parallel pipelined 2-D FFT configured as two tasks, each on four processors and operating on arrays of size 8×8. Shading indicates array elements mapped to processor 0 in task 0 and in task 1. Note that the arrays `A` and `B` are mapped to disjoint sets of processors.

MPI provides a set of functions, datatypes, and protocols for exchanging data among and otherwise coordinating the execution of multiple tasks; a "binding" defines the syntax used for MPI functions and datatypes in a particular language. Previous MPI implementations have supported bindings only for the sequential languages C and Fortran 77 [9]. 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.

Figure 3 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

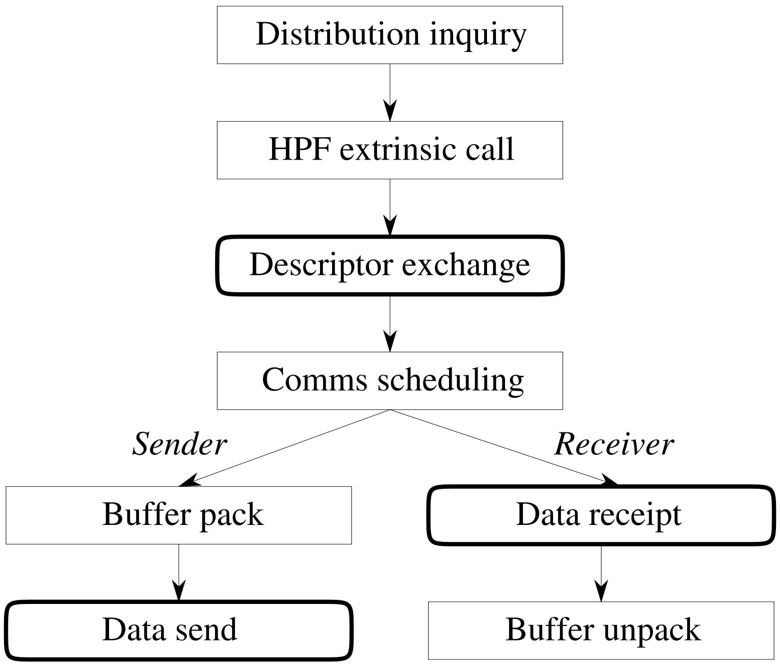

Figure 4: The steps executed during an HPF/MPI communication function. The rounded boxes distinguish the steps involving communication. The sending and receiving sides differ only in the last two steps.

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 Figure 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, 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

An HPF/MPI implementation must address a variety of HPF- and MPI-specific issues, particularly at the interface between HPF and MPI, as well as general issues relating to the transfer of distributed data structures among process groups. We briefly describe the techniques that we have developed to address these issues. We focus on point-to-point communication and consider just one of several possible implementation approaches, namely that illustrated in Figure 4. In the following, we describe each of the six steps involved in this figure, looking at the actions performed during a send operation.

1. *Distribution inquiry.* Standard HPF inquiry intrinsics are called to determine the distribution of the array that is to be communicated.

2. *HPF extrinsic call.* The communication operation is invoked as an HPF extrinsic call to a procedure in the HPF/MPI library. The procedure is invoked in "local" mode, meaning that a separate thread of control executes on each processor on which the invoking task is running [14].

3. *Descriptor exchange.* Sending processors exchange distribution information with receiving processors. In general, each sending processor need communicate only with a subset of the receiving processors.

4. *Communications schedule.* Sending processors use the distribution information obtained in Step 3 to determine which subsections of the input array should be sent to each receiving processor.

5. *Buffer pack.* The schedule information computed in Step 4 is used to pack the array elements required by a particular receiving processor into a buffer. (Steps 5 and 6 are repeated once for each processor to which data must be sent.)

6. *Data send.* The contents of the buffer packed in Step 5 are sent to the appropriate receiving processor.

We have implemented a prototype HPF/MPI library using these techniques. This library supports a subset of MPI's point-to-point communication functions and operates with `pghpf` (version 2.0), a commercial HPF compiler developed by the Portland Group, Inc. Our library requires minor modifications to the `pghpf` runtime system to create the initial set of tasks when a computation is started and to provide information about which tasks execute on which processors. The communication schedules required in Step 4 are computed with algorithms based on the FALLS (FAmiLy of Line Segments) representation of Ramaswamy and Banerjee [17]. These algorithms incorporate efficient and general techniques for computing the minimal sequence of communication operations required to perform a redistribution. Note that while the implementation strategy of Figure 4 is efficient for typical multicomputers, other strategies are possible and may perform better in some situations. For example, in a low-latency network it may be useful to pipeline communications, while in a low-connectivity network it may be worthwhile to gather all data to one node, perform the transfer by using a single message, and then scatter from one receiving node.

The techniques just described can be refined and optimized in various ways to improve performance in specific situations. For example, MPI provides functions `MPI_Send_init` and `MPI_Recv_init` to define what are called *persistent requests*; once defined, these requests can be executed repeatedly using a third function, `MPI_Start`. As illustrated in Figure 5, MPI programmers can use these functions to indicate that the same data transfer will be performed many times. An HPF/MPI implementation of these calls can compute communication schedule information once (within the `Init` functions) and subsequently reuse this information (within `MPI_Start`) so that costs associated with Steps 1, 3, and 4 are amortized over many communications.

```
!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
```

Figure 5: 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`.

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 Figure 6, exchanges repeatedly a 2-D array of fixed size between two tasks, where in each communication the array is distributed `(BLOCK,*)` in the sender and `(*,BLOCK)` in the receiver. We run this program for arrays of varying size and for varying numbers of processors within each task, allowing us to measure the total time per one-way communication in different situations. We also measure the time spent in the six steps illustrated in Figure 4. 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.

Figure 7 shows our results. In studying these results, we first note that for small problem size (N), cost increases with number of processors (P), while for large N, costs 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.

It is useful to relate achieved performance to the sources of the various cost components. Steps 1, 3, and 4 are associated with determining how to perform a communication and can be avoided if persistent communications are used. These three components are shown uppermost in each bar, which in most cases allows us to distinguish the costs for nonpersistent and persistent communication. We note, however, that the time for Step 3 (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 are an artifact of the experimental protocol, not a sign of inefficiency in the implementation of descriptor exchange.

Step 2 (HPF extrinsic call) represents the costs associated with the extrinsic interface. This component represents a fixed cost for multiple subroutine calls, plus a per-word overhead incurred by the use of the HPF extrinsic subroutine interface. Because the `pghpf` compiler uses a specialized internal representation for arrays, it typically must copy a distributed array

```
!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
```

Figure 6: 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).

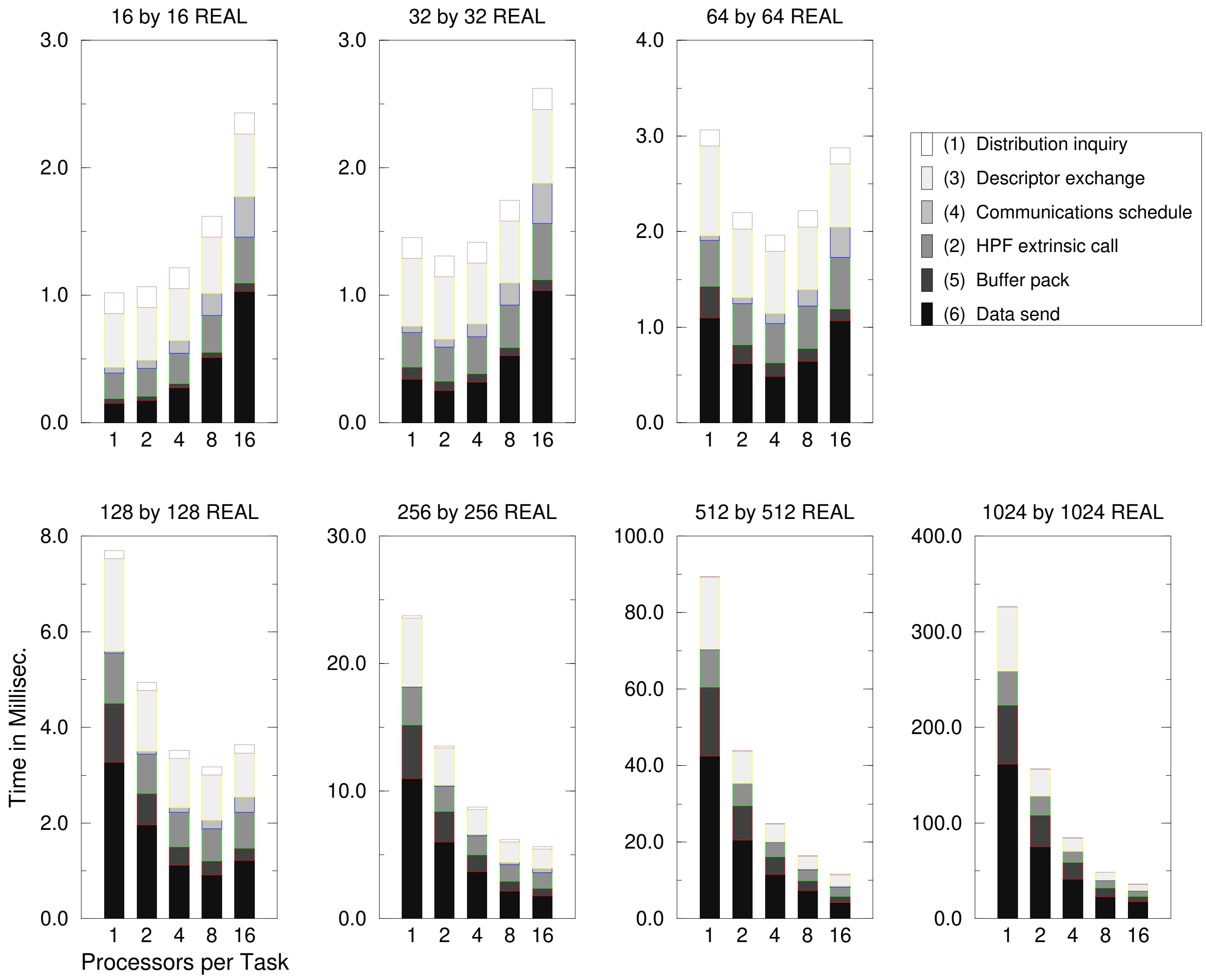

Figure 7: Time required for a one-way HPF/MPI point-to-point communication on an IBM SP2, for various array sizes and numbers of processors in the sending and receiving tasks.

passed as an extrinsic procedure's input argument into one contiguous region, to preserve the property of array element *sequence association* assumed within Fortran 77; return array arguments are similarly copied upon extrinsic subroutine return. Both the subroutine calls and the copying represent overhead that could, in principle, be avoided by a tighter coupling of HPF and the MPI library. (For example, buffer packing and unpacking operations could be performed directly on the `pghpf` internal array representation.) When $P{=}1$ and $N{=}16$ (1 KB data), Step 2 costs 200 microseconds; when $P{=}1$ and $N{=}1024$ (4 MB data), the cost is 33 milliseconds. These data suggest a fixed cost of roughly 200 microseconds and an incremental cost of 0.008 microseconds/byte (106 MB/sec bandwidth).

Step 5 (Buffer pack/unpack) corresponds to the costs incurred when transferring data between potentially noncontiguous locations in an array and a communication buffer. Our implementation performs these transfers explicitly in all cases; optimized implementations might be able to avoid this extra copying for some distributions on some platforms. As the amount of copying performed in Step 5 appears to be equivalent to that performed in the extrinsic interface, we might expect Steps 2 and 5 to have similar costs. In practice, we find that for large messages Step 2 runs at about 106 MB/sec while Step 5 achieves only 58 MB/sec. We are currently investigating the reason for this difference, which we suspect is due to more highly optimized copying routines in `pghpf`.

The final component is the actual communication. Since we always use a direct communication structure, we expect cost to be roughly $Pt_s + (N^2/P)t_w$, where P is the number of processors per task, t_s is the per-message startup cost, and t_w is the per-word data transfer time. The experimental data fit this model reasonably well.

Overall, we see that the persistent communication optimization can make a large difference for small N (up to 40–60 percent, depending on P) but has progressively less impact as N increases, always accounting for less than 25 percent for $N{\geq}256$. Extrinsic call and buffer pack/unpack overheads vary significantly with N and P, peaking at around 50 percent of the time remaining once the persistent communication optimization has been applied. For $N{=}1024$ and $P{=}1$, we achieve a transfer rate of 12.8 MB/sec without the persistent communication optimization; the low-level MPICH library on which our HPF/MPI library is based achieves 30 MB/sec in this situation.

In summary, the microbenchmark results show that the persistent communication optimization provides significant benefits, particularly for 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 there is considerable benefit to be gained from a tighter coupling of HPF and MPI implementations.

6 Larger Programs

We also studied the performance of HPF/MPI implementations of 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.

2-D FFT. 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 HPF/MPI code is executed

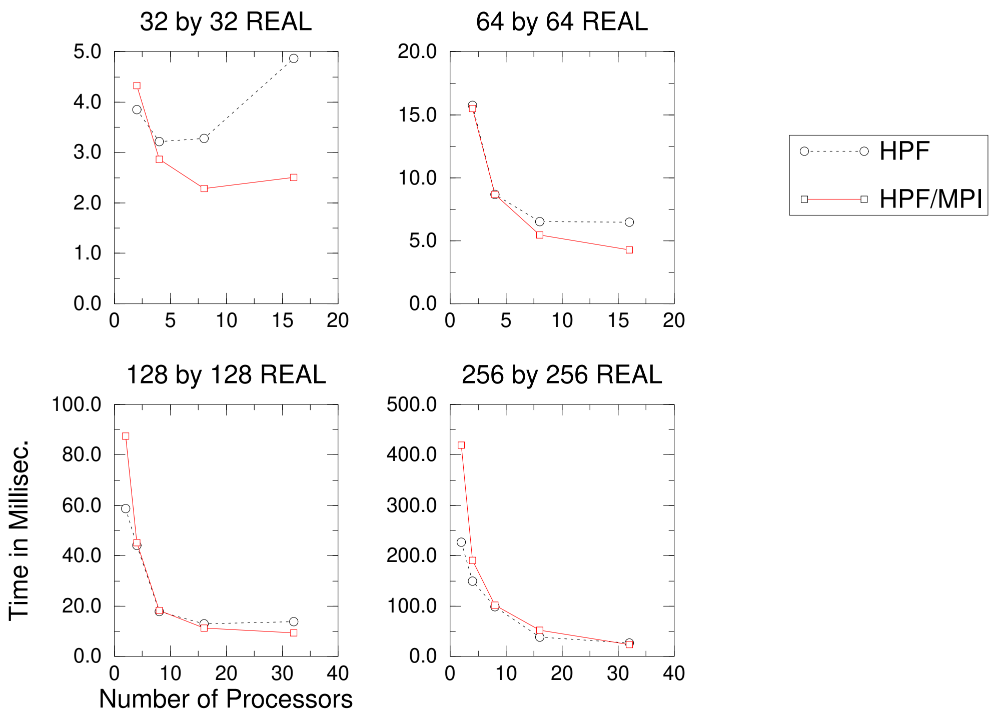

Figure 8: Execution time per input array for HPF and HPF/MPI implementations of the 2-D FFT application, as a function of the number of processors. Results are given for different problem sizes.

as a pipeline of two tasks, with an equal number of processors assigned to each task. Figure 8 presents our results, which are performed for a number of images large enough to render pipeline startup and shutdown costs insignificant. As expected, the HPF/MPI program is faster than the HPF code for larger numbers of processors and smaller problems.

Convolution. Convolution is a standard technique used to extract feature information from images [4, 19]. It involves two 2-D FFTs, an elementwise multiplication, and an inverse 2-D FFT (Figure 9) and is applied to two streams of input images to generate a single output stream. A data-parallel convolution algorithm performs the steps illustrated in Figure 9 in sequence for each image, while a pipelined algorithm can execute each block in Figure 9 as a

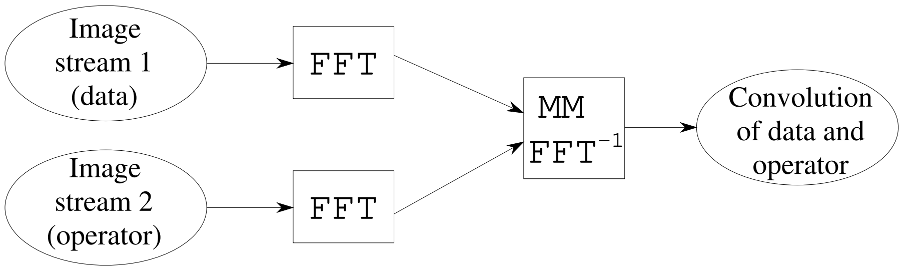

Figure 9: Convolution algorithm structure. Two image streams are passed through forward FFTs and then to a pointwise matrix multiplication (MM) and inverse FFT.

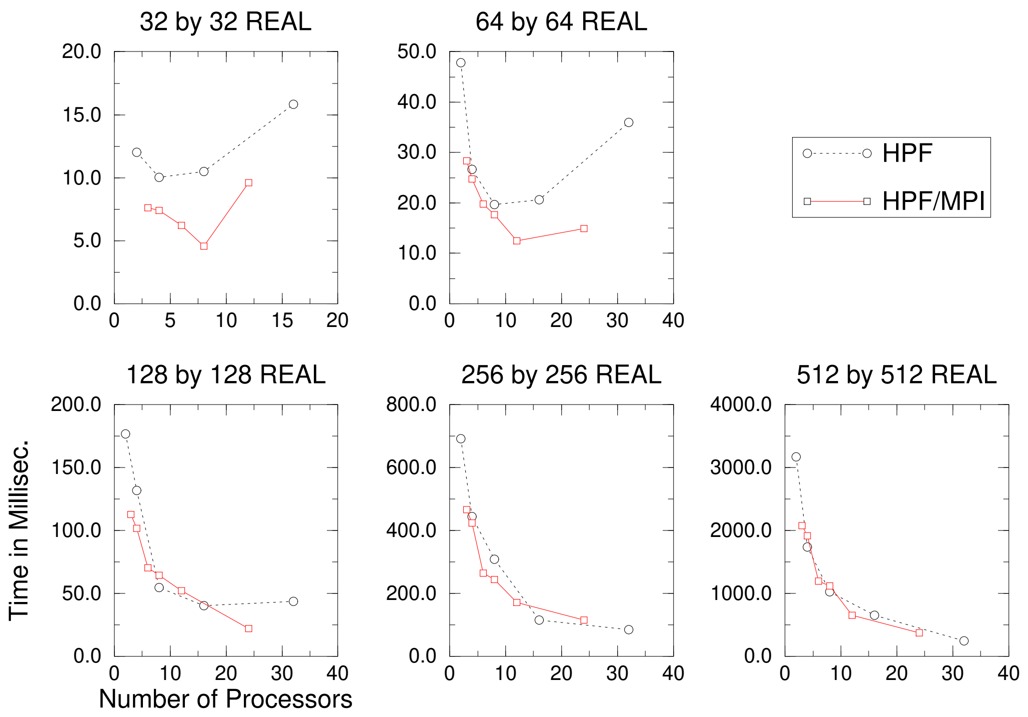

Figure 10: Execution time per input array for HPF and HPF/MPI implementations of convolution, as a function of the number of processors. Results are given for different problem sizes.

separate task. As in the 2-D FFT, this pipeline structure can improve performance by reducing the number of messages. (It is also possible to exploit pipelining within each FFT, to increase parallelism further. We do not consider this option here.) Figure 10 shows our results. Once again, we see that the HPF/MPI version is often significantly faster than the pure HPF version.

Multiblock. Multiblock codes decompose a complex geometry into multiple simpler blocks [21]. A solver is run within each block, and boundary data is 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 [7]. Figure 11 shows the three-block geometry used in our experiments, and an intermediate solution computed on this geometry. 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/MPI version, processors are allocated to blocks in proportion to their size.) Figure 12 shows our results. The HPF/MPI program is always faster than the pure HPF program.

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

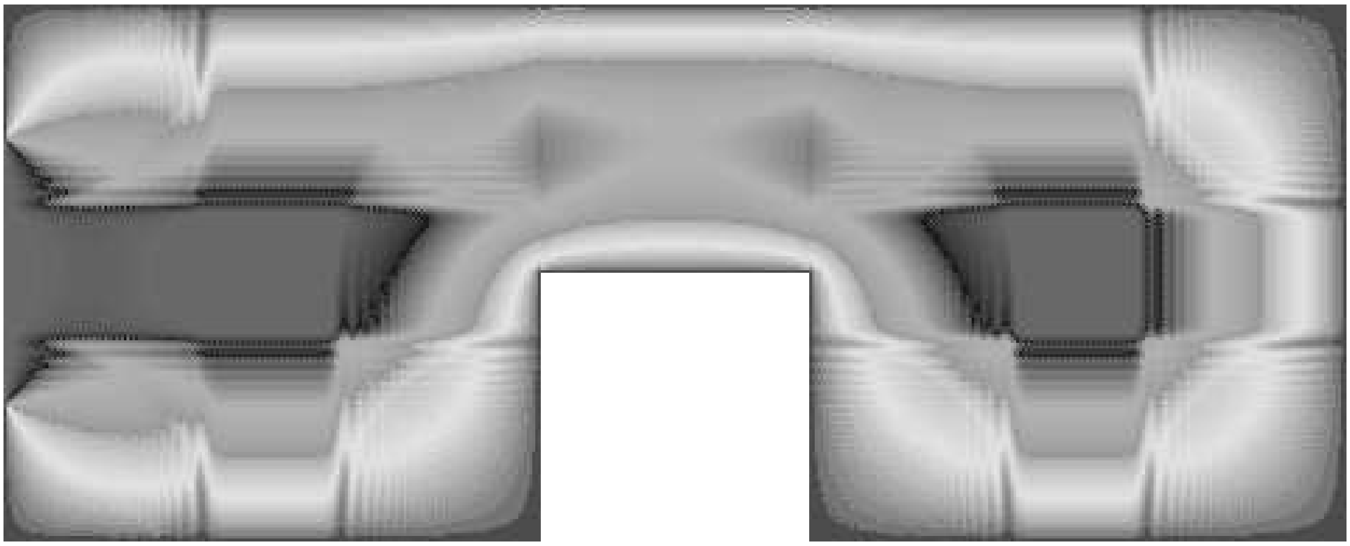

Figure 11: The three-block geometry used to evaluate the performance of the multiblock code. The three blocks shown in this figure have size 192×192, 96×96, and 192×192.

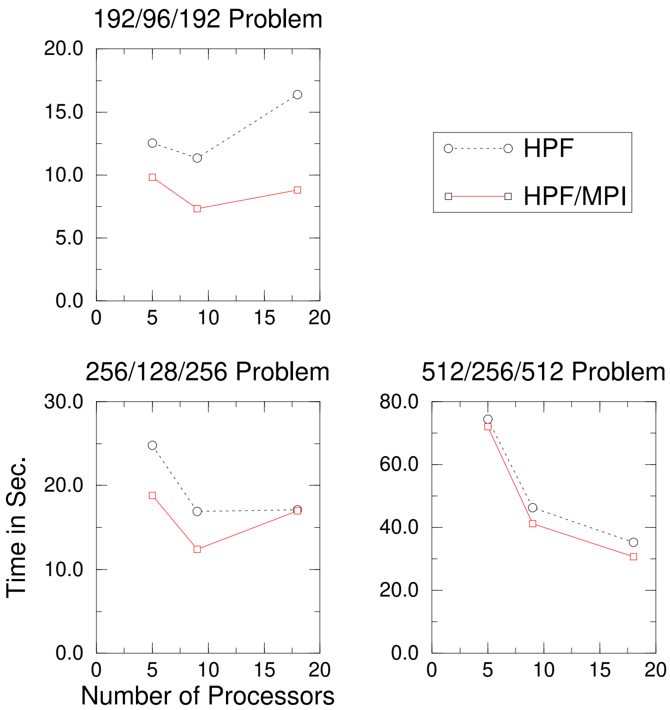

Figure 12: Execution time for HPF and HPF/MPI implementations of the multiblock code, as a function of the number of processors.

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 remain. The extent to which these overheads can be avoided by a tighter coupling between HPF/MPI and `pghpf`, by refining the HPF extrinsic interface or by using compiler-derived information to select specialized communication functions, are topics for future research.

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.

Acknowledgments

We are grateful to the Portland Group, Inc., for making their HPF compiler and runtime system available to us for this research, and to Shankar Ramaswamy and Prith Banerjee for allowing us to use their implementation of the FALLS algorithm. The multiblock Poisson solver is based on a code supplied by Scott Baden and Stephen Fink. We have enjoyed stimulating discussions on these topics with Chuck Koelbel and Rob Schreiber. This work was supported by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

References

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

[2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[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, Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for pipeline computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, 1994.

[5] I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 293–300. IEEE Computer Society Press, 1994.

[6] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1990.

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

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

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Preprint MCS-P567-0296, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Processing with the Message-Passing Interface.* The MIT Press, Cambridge, Mass., 1994.

[11] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(2):16–26, Fall 1994.

[12] W. Hillis and G. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

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

[14] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[15] 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.

[16] 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, Ill., 1994.

[17] Shankar Ramaswamy and Prithviraj Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, Va., February 1995.

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

[19] A. Rosenfeld and A. Kak. *Digital Picture Processing.* Academic Press, New York, 1976.

[20] B. Seevers, M. Quinn, and P. Hatcher. A parallel programming environment supporting data-parallel modules. *International Journal of Parallel Programming*, 21(5), October 1992.

[21] V. N. Vatsa, M. D. Sanetrik, and E. B. Parlette. Development of a flexible and efficient multigrid-based muliblock flow solver; AIAA-93-0677. In *Proc. 31st Aerospace Sciences Meeting and Exhibit*, January 1993.

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