ANL/MCS-P--597-0596
CONF-9607124--5

MPI as a Coordination Layer for Communicating HPF Tasks

Ian T. Foster[*]
David R. Kohr, Jr.
Mathematics and Computer Science Div.
Argonne National Laboratory
Argonne, IL 60439
{foster,kohr}@mcs.anl.gov

Rakesh Krishnaiyer
Dept. of Computer and Information Science
Alok Choudhary
Dept. of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244
{rakesh,choudhar}@cat.syr.edu

Abstract

Data-parallel languages such as High Performance Fortran (HPF) present a simple execution model in which a single thread of control performs high-level operations on distributed arrays. These languages can greatly ease the development of parallel programs. Yet there are large classes of applications for which a mixture of task and data parallelism is most appropriate. Such applications can be structured as collections of data-parallel tasks that communicate by using explicit message passing. Because the Message Passing Interface (MPI) defines standardized, familiar mechanisms for this communication model, we propose that HPF tasks communicate by making calls to a coordination library that provides an HPF binding for MPI. The semantics of a communication interface for sequential languages can be ambiguous when the interface is invoked from a parallel language; we show how these ambiguities can be resolved by describing one possible HPF binding for MPI. We then present the design of a library that implements this binding, discuss issues that influenced our design decisions, and evaluate the performance of a prototype HPF/MPI library using a communications microbenchmark and application kernel. Finally, we discuss how MPI features might be incorporated into our design framework.

1. Introduction

Message-passing libraries such as the Message Passing Interface (MPI) provide programmers with a high degree of control over the mapping of a parallel program's tasks to processors, and over inter-processor communications [5]. However, this control comes at a high price: programmers must explicitly manage all details relating to parallelism, such as synchronization and data transfer. In contrast, data-parallel languages such as High Performance Fortran (HPF) provide a simple programming model in which all processors execute a single, logical thread of control that performs high-level operations on distributed arrays; many tedious details are managed automatically by the compiler [7].

1.1. Limitations of data parallelism

While data-parallel languages such as HPF can greatly ease development of concise solutions to many parallel programming problems, the rate of improvement of speedup of many data-parallel programs diminishes sharply as more processors are used to execute a program. This is typically due to increased communication overhead. Alternatively, one may say that *parallel efficiency*, or the ratio of speedup to processors, decreases as the number of processors increases. Figure 1 depicts an abstract example of this phenomenon. Classes of applications that exhibit this effect most markedly include those that perform a number of heterogeneous processing steps (such as pipeline codes and multidisciplinary simulations) and those that operate on irregularly-structured data (such as multiblock codes).

Fortunately, many such programs can be decomposed into independent data-parallel tasks that can execute in parallel on a subset of the available processors at higher parallel efficiency than the original program running on all processors [2, 6]. For example, suppose the program of Figure 1 can be reformulated as a pair of communicating data-parallel tasks that each run on $\frac{P}{2}$ processors with a parallel efficiency of 90% (as did the

[*] To whom correspondence should be addressed.

The submitted manuscript has been authored by a contractor of the U. S. Government under contract No. W-31-109-ENG-38. Accordingly, the U. S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U. S. Government purposes.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

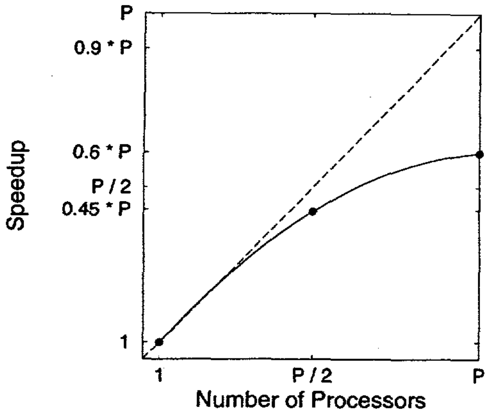

Figure 1. A plot of speedup versus number of processors for an application that exhibits diminishing parallel efficiency.

original program). When this mixed task/data-parallel version executes on all P processors, it can maintain a parallel efficiency of 90%, a significant improvement over the 60% of the purely data-parallel version.

Though this simple analysis neglects the additional inter-task communication incurred by the task-parallel version, in practice this overhead often is dominated by the improvement in each task's parallel efficiency. Moreover, in many pipeline applications it is desirable to optimize not the time to process a single dataset (the pipeline *latency*), but rather the number of datasets processed per unit time (the *throughput*). Throughput is bounded not by the time to complete all stages, but rather by the processing rate of the slowest stage. Therefore, even if communication overhead causes the latency of a pipelined version to rise above that of a purely data-parallel version, so that the speedup of the pipeline at processing one dataset is actually lower, the pipeline may still be preferable because its throughput is higher [1].

1.2. MPI in an HPF context

Because HPF is a powerful, high-level notation for expressing data-parallel computations, while MPI facilitates precise control over task mapping and inter-task communication, we propose the use of an HPF binding for MPI as a coordination layer for coupling together data-parallel tasks to construct mixed task/data-parallel programs. However, the semantics of a standard such as MPI that is intended for sequential languages are not entirely clear when its mechanisms are invoked from a parallel language. For example, a "process" in MPI is assumed to be an independent thread of control executing on a single processor. This is ambiguous when applied to the execution model of HPF, where one logical thread of control is replicated across many physical processors. Similarly, data structures in MPI are assumed to reside within a single address space, yet a fundamental premise of HPF is that arrays can be distributed across multiple address spaces.

```
Producer (task 0):
   !HPF$ processors prod_procs(4)
         real A(8, 8)
   !HPF$ distribute A(BLOCK, *) onto prod_procs
         do i = 1, N
           call produce_data(A)
           call MPI_Send(A, 8*8, MPI_REAL, 1, 99,
      &                  MPI_COMM_WORLD, ierr)
         end do

Consumer (task 1):
   !HPF$ processors cons_procs(2)
         real B(8, 8)
   !HPF$ distribute B(*, BLOCK) onto cons_procs
         do i = 1, N
           call MPI_Recv(A, 8*8, MPI_REAL, 0, 99,
      &                  MPI_COMM_WORLD, status,
      &                  ierr)
           call consume_data(B)
         end do
```

Figure 2. Producer-consumer example written using HPF/MPI.

Our definition of an HPF binding for MPI attempts to resolve these difficulties. In an HPF/MPI program, each task constitutes an independent HPF program in which one logical thread of control operates on arrays distributed across a statically-defined set of processors. At the same time, each task is also one logical process in an MPI computation. Therefore, tasks may communicate and synchronize with one another by calling standard MPI routines for point-to-point transfer and collective operations. The combination of the semantics of our binding and the implicit nature of parallelism in HPF yields the following helpful consequence: when reading an HPF/MPI program one may ignore the HPF directives and treat the remainder as a parallel Fortran 90 program containing explicit message-passing calls.

We use a very simple producer-consumer example to illustrate the usage of the HPF binding for MPI; Figure 2 shows the source code for the example. The producer task calls the function produce_data, which

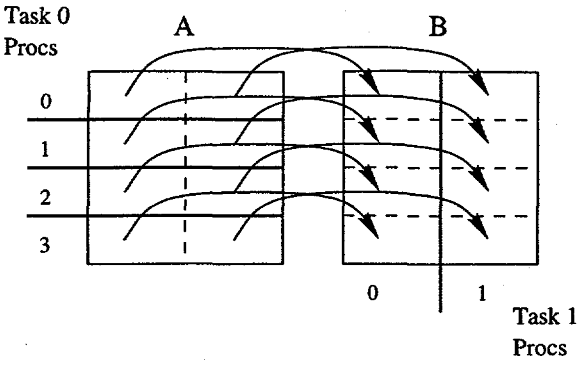

Figure 3. Movement of distributed data from producer to consumer.

performs a series of data-parallel operations on the array A using four processors. Then the producer calls MPI_Send to transmit the contents of A to the consumer, in a message with a tag value of 99. The consumer receives this data into an array B, using MPI_Recv. Finally, the consumer processes the data in parallel on its two processors by calling consume_data.

What distinguishes this example from an ordinary sequential MPI program is that each of the two logical MPI processes is an HPF task executing on several processors. Hence the source array being transferred via MPI calls is actually distributed across the processors of task 0, and the message destination is distributed across those of task 1. Figure 3 depicts the complex pattern of data movement from source to destination required to perform this transfer. Yet from the programmer's perspective, one invokes just a single transfer operation; all the complexity is encapsulated in the HPF/MPI library.

The example of Figure 2 does not show how the two tasks were connected together in a pipeline. This can be achieved in two ways:

1. During execution, tasks may invoke the inquiry function MPI_Comm_rank to determine their identity, and perform conditional processing based on the returned value. (This is similar to the operation of SPMD programs.)

2. The startup mechanism of an HPF/MPI implementation must permit definition of the size of each task. If the startup mechanism also lets the user specify different programs to be executed by different tasks, then a collection of separately-compiled executables may be combined into a single HPF/MPI computation. (Many implementations of sequential MPI permit this.)

In the next section, we present the design of a library that implements a subset binding of MPI, based on the ideas just presented. In Section 3, we evaluate the performance of a prototype HPF/MPI library, and determine the sources of overheads that affect its performance. Section 4 contains a discussion of promising techniques for extending our library to include additional MPI features. Finally, in Section 5 we compare our techniques for introducing task parallelism into data-parallel languages with other approaches, state our conclusions regarding the effectiveness of our approach, and suggest directions for future work.

2. An Implementation Strategy

We have designed and implemented a subset of an HPF binding for MPI that provides the communication operations described above. Because the implementation of all of MPI is a daunting task, we have restricted our efforts to a small subset so that we can focus on analyzing and understanding design and performance issues. Our HPF/MPI implementation operates with the commercial HPF compiler pghpf, developed by the Portland Group, Inc. [9]

The design of our HPF/MPI library was guided from the outset by several underlying assumptions and objectives, including the following:

- The primary target platforms on which we would run HPF/MPI applications would be distributed-memory multicomputers.

- We wished to maintain a high degree of portability across hardware and software platforms, including across different HPF compilation systems.

- The library should achieve good performance for communication patterns typical of the sorts of mixed task/data-parallel applications we wished to support.

- When users express optimization hints through MPI facilities (such as the fact that a particular communication pattern is repeated many times), HPF/MPI should be able to exploit these opportunities.

- It should be possible to build upon the subset library to extend it into a full implementation of all of the MPI standard.

These guiding principles carry with them a number of important consequences for our design. For example, the characteristics of our intended target platforms imply that to achieve high transfer bandwidth for large arrays, during communication we should try to utilize the high connectivity of the target's network by performing multiple transfers in parallel. As a result, we have developed a design based on a *parallel strategy* (described below).

Furthermore, as a result of our desire for portability, we chose a sequential implementation of MPI as the underlying communication substrate, because it is available on many multicomputers and utilized by many HPF compilers. We note, however, that HPF/MPI can be layered atop other communication substrates. In Section 4, we discuss how functionality beyond that provided by MPI could aid in extending our subset library.

Many of the applications we wish to support require low latency for certain communications which are repeated frequently [1]. MPI includes the functions `MPI_Send_init` and `MPI_Recv_init` for defining *persistent requests* for sends and receives; persistent requests allow an implementation to recognize and optimize such repeated operations. Therefore we selected persistent requests as the first MPI optimization facility to add to our library. The difficulty of incorporating this feature into the library also served as a measure of the modularity of our design: the more modular the design, the easier it will be to extend HPF/MPI to support the entire MPI standard.

2.1. Details of the implementation

When an HPF task invokes an HPF/MPI communication function, the library takes a number of actions to effect the data transfer. Here we examine the sequence of steps taken by the producer (task 0) in Figure 2 as it calls `MPI_Send` to transfer distributed source array `A` to destination array `B` in task 1. The steps are as follows:

1. *Distribution inquiry*: Standard Fortran 90 and HPF intrinsic inquiry functions are called to create an array descriptor for `A` that specifies its size and distribution.

2. *HPF extrinsic call*: A C language data transmission routine `mpi_send_c` is invoked. Because the routine is not written in HPF, it must be invoked in *local mode*: for the duration of the call, each processor in task 0 has a separate thread of control (SPMD-style execution) rather than the single thread of control implied by HPF.

3. *Array descriptor exchange*: Processors in task 0 join in a collective operation with those of task 1. This operation has the effect of broadcasting the array descriptor of `A` to all processors in task 1, and that of `B` to all processors in task 0. (We exploit the fact that each descriptor is initially present on all processors of one task by implementing this operation using a set of point-to-point transfers; this is typically more efficient than a broadcast.)

4. *Communications scheduling*: Using the array descriptors for `A` and `B`, each processor of task 0 computes a *communications schedule*, that is, the sets of elements of its local portion of `A` that must be sent to each processor of task 1. The schedule is computed by algorithms based on the FALLS representation of Ramaswamy and Banerjee [8].

5. *Transfer buffer packing*: The elements to be sent to a single processor of task 1 are packed (gathered) into a single contiguous *transfer buffer*.

6. *Data transmission*: The contents of the transfer buffer prepared in Step 5 are transmitted to the receiving process using point-to-point operations of the underlying communication substrate.

Note that Steps 5 and 6 are performed by a sending processor once for each receiving processor that requires array elements from the sender.

In the case of a task receiving data using `MPI_Recv` (such as the consumer task of Figure 2), the sequence of steps is essentially the same through the end of Step 4. In Step 5, each processor receives data from a sending processor into a transfer buffer, using the sequential version of `MPI_Recv`. Finally, in Step 6 the contents of the transfer buffer are unpacked (scattered) to their final locations in the destination array. As in the case of a sending task, Steps 5 and 6 are repeated once for every sending process from which elements must be received. The iteration ordering for each receiver over its set of senders is chosen to match the iteration ordering for senders over their receivers, so that the send and receive operations comprising a data transfer match correctly.

When a task creates a persistent request for a send or receive using `MPI_Send_init` or `MPI_Recv_init`, its processors execute Steps 1 through 4 of the sequence presented above, and the resultant communications schedule is cached in an `MPI_Request` object. When the request is subsequently executed using `MPI_Start`, this communications schedule is used to perform Steps 5 and 6. Therefore the delay incurred by descriptor exchange and the processing overhead of communications scheduling can be amortized over many operations.

2.2. Properties of the implementation

To obtain the best performance, it is important that transfers between different senders and receivers proceed in parallel. This implies that two senders should not try to send to the same receiver at the same time. As transfers are performed iteratively by each sender, the parallelization of transfers depends on the iteration ordering of each sender over its set of receivers, which is selected by the FALLS-based algorithms. Transfers generally proceed in parallel if both of the following conditions are met:

1. There are at least as many receivers as senders. This condition depends on the sizes of the sending and receiving tasks.

2. All senders possess the same set of receivers. This condition holds for most common redistributions.

Because there is no synchronization between senders as they iterate over receivers, it is possible for one sender to overtake another, with the result that both send to the same receiver at the same time. The receiver then becomes a hotspot, and parallelism is reduced. However, if each transfer is of roughly the same size (as is the case for many common redistributions), this is unlikely to occur.

As noted above, portability across platforms and HPF compilation systems was one of our major goals. To this end, we defined a simple link-level interface which we believe permits our library to work with the run-time system of any HPF compiler that uses MPI as its communication substrate. The key to portability is that the interface does not require access to the HPF system's internal data structures. When we tested this interface with **pghpf**, only minimal modifications to the source code of **pghpf**'s run-time system were necessary.

High-quality implementations of HPF may store the local portion of a distributed array in a non-contiguous format denoted by an internal run-time array descriptor: this permits optimization of compiler-synthesized communication (e.g. by padding arrays with "ghost elements"), and efficient operation on array subsections. To avoid dependence on **pghpf**'s array descriptor format, we explicitly declare HPF/MPI communication routines to be *extrinsics*, or routines not written in HPF. This declaration causes array arguments to be copied into a contiguous temporary array before entering the extrinsic, and copied back from the temporary upon return. The temporary array possesses the property of *sequence association* currently required by the HPF/MPI library. As a result, the library remains portable across HPF compilation systems. Unfortunately, this portability comes at a cost in performance; see Section 3.

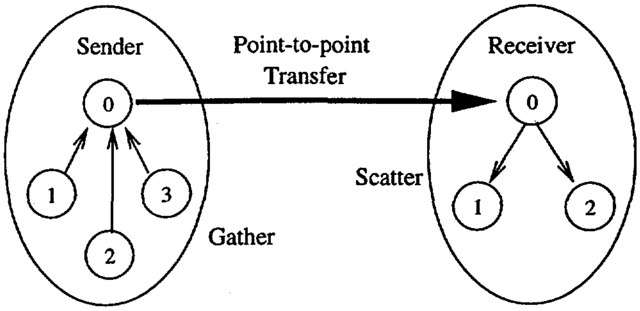

Figure 4. Communication operations performed for the centralized strategy. Ovals are tasks, small circles are individual processors.

2.3. An alternative strategy

The parallel strategy presented above involves all processors in simultaneous transfers of sections of the array. This reduces transfer time for large arrays at the expense of requiring all array descriptors to be distributed to all processors, which can increase total transfer time for small arrays. Therefore, for transfers of small arrays, or when executing on networks with low connectivity where parallel transfers are not appropriate, we have developed an alternative design based on a *centralized strategy*. This design does not require global distribution of descriptors and does not attempt parallel transfers. This scheme is depicted in Figure 4; it operates as follows:

1. The entire array is gathered at a single sending processor using a sequential MPI collective operation.

2. The entire array is transmitted to a single receiving processor using sequential MPI functions.

3. The array is scattered to all receivers using a sequential MPI collective operation.

3. Performance Results

In this section we evaluate the performance of an implementation of a subset of the HPF binding of MPI that relies on the parallel strategy. We use a standard synthetic benchmark to identify sources of overhead in the implementation and to investigate the effectiveness of the optimization for persistent operations. We suggest techniques for reducing the overheads revealed by these measurements. Then, we compare the execution times of pure HPF and HPF/MPI versions of a 2D FFT application kernel, to judge the utility of HPF/MPI for accelerating real data-parallel programs.

All experiments were performed on Argonne's IBM SP system, which consists of 128 Power 1 processors linked by an SP2 interconnection network. The underlying sequential MPI library was MPICH [4]. All HPF programs were compiled with pghpf, using what we determined to be the most effective optimization switches.

3.1. Communication performance

To evaluate the performance of our library at transferring distributed arrays between tasks, we use a data-parallel variant of the standard "ping-pong" communication benchmark. This program consists of two tasks with equal numbers of processors that alternately send and receive 2D arrays of a fixed size a large number of times. The arrays are distributed (BLOCK, *) on the sending side and (*, BLOCK) on the receiving side. Hence a worst-case redistribution is performed during each transfer, as each sending processor must communicate with all receivers.

The performance achieved by HPF/MPI depends in part on the performance of the underlying sequential MPI implementation. There is a simple, widely-used model that accurately characterizes the behavior of point-to-point transfer operations by many message-passing libraries running on multicomputers. This model assumes that for a message N bytes long, the time T_N to transfer the message between two processors is governed by the equation

$$T_N = t_s + N \times t_b$$

where t_s is the communication startup time, or latency, and t_b is the time to transfer one byte of the message (inversely related to the bandwidth). For the MPICH layer used in our experiments, we measured a latency t_s of 87.9 μsec and a per-byte cost t_b of 0.0326 μsec, which corresponds to a bandwidth of 30.7 Mbytes/sec.

Figure 5 shows the time measured using the ping-pong benchmark for one-way non-persistent and persistent transfers of small and large messages, with varying numbers of processors P per task. In general, for short messages we find that transfer time increases with increasing P, while transfer time decreases as P rises for large messages. In terms of the above model, for small non-persistent transfers, t_s is $85.3 \times P + 1290$ μsec; for persistent transfers, t_s is $60.0 \times P + 827$ μsec. Both are roughly proportional to the latency of the sequential MPI substrate. (These values were determined using a least-squares fit.) For large messages, the per-byte cost is 0.081 μsec, which yields a peak bandwidth of 12.4 Mbytes/sec. The persistent optimization decreases transfer time by 26–32% for small messages, depending on P, while for large messages it has negligible effect.

By examining the time spent in each of the six processing steps of our design, we can often identify the sources of overheads that contribute to the transfer time. Such a breakdown of the total time is represented by the shaded regions within each vertical bar of Figure 5. The time for each step appearing here is the maximum among all processors (the variance across processors was low). Since we are interested in the end-to-end time to transfer data from a sender to a receiver, the times for corresponding steps for sending and receiving messages have been summed together.

From this breakdown, we find that distribution inquiry (Step 1) has a small, fixed cost, never more than 10% of the total. The time to compute a communication schedule (Step 4) also has a modest cost, though it rises with P. This is because the FALLS-based algorithms require time proportional to the larger of the number of senders or receivers. For small messages, descriptor exchange (Step 4) requires about 500 μsec, which is 15–30% of the total (depending on P). For large messages, a long time is spent in this step (up to 20 millisec, or 22%). This phenomenon is not due to a message-size-dependent cost for exchanging descriptors, but rather because of synchronization delays resulting from a load imbalance: after a sender completes transmission of a message, it immediately initiates a receive, and waits at the descriptor exchange step—a synchronization point—while the receiver finishes receiving and unpacking data messages. All three of these steps are skipped when persistent communications are performed; however, for large messages most of the time spent in descriptor exchange shifts to data transmission (Step 6), which is the other point of synchronization during a transfer.

The cost of the HPF extrinsic call (Step 2) includes both a fixed overhead of about 200 μsec (mostly subroutine call overhead) and a per-byte cost for argument copying, as noted in the previous section. As a result, this step takes 10-20% of the total time. Presumably much of this overhead would disappear if our library were able to operate directly on pghpf's internal representation of arrays, so that it would not need to be invoked using the HPF extrinsic mechanism.

Buffer packing and unpacking (Step 5) includes a

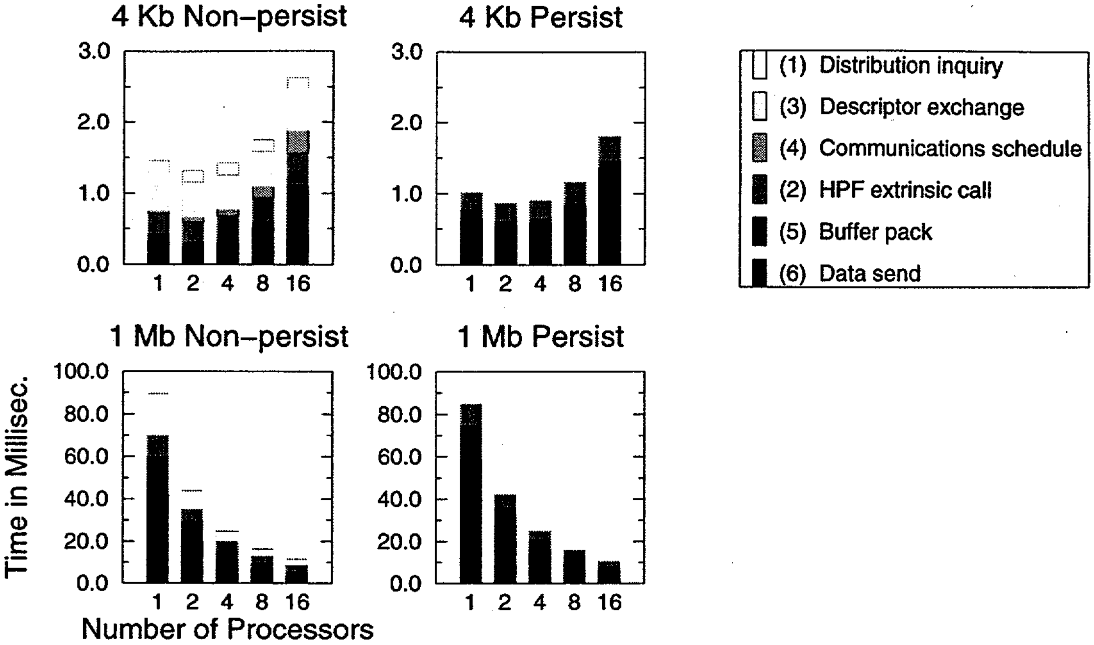

Figure 5. One-way message transfer times for small (4 kilobyte) and large (1 megabyte) messages, using non-persistent and persistent operations. The time spent in different processing steps is denoted by the shaded regions within each vertical bar.

per-byte cost that causes this step to consume about 20% of the total time for large messages. The processing in this step is a kind of scatter-gather operation. Because data is always copied to an intermediate buffer before being transferred to its final location, we will refer to this operation as an *indirect* scatter-gather. The user-defined datatype facilities of MPI make it possible to specify a *direct* scatter-gather, in which data can be transferred directly between the network interface and non-contiguous locations within a program's data structures, without buffering. However, not all MPI implementations can actually perform this direct transfer. Therefore, in principle it should be possible for HPF/MPI to specify a direct scatter-gather in this step, which could result in a large reduction in overhead on some platforms. However, for many redistributions the complexity of the required MPI datatypes is quite high. (The creation of these datatypes is even more complex if one performs the direct scatter-gather on the HPF run-time system's non-contiguous internal representation of arrays.) Hence modifying the library to perform a direct scatter-gather on general distributions would require extensive enhancements to the FALLS-based scheduling algorithms, though there are common, simpler redistributions that are easier to handle.

The time spent performing data transmission (Step 6) varies in a predictable manner with N and P. For small messages, the time is roughly proportional to P; this is to be expected, as each processor must send and receive P messages. The constant of proportionality is about the same as the value of t_s for the underlying sequential MPI library. For large messages, the time is proportional to the amount of data per processor (hence inversely related to P). The achieved bandwidth per processor ranges from 16 to 26 Mbytes/sec, always at least half that of the underlying MPI substrate. The bandwidth generally drops with increasing P. We suspect that this decrease in bandwidth is due to the domination of startup overhead as the amount of data per processor drops, as well as synchronization delays, but further investigation is required.

3.2. Application performance

Synthetic communication microbenchmarks such as the ping-pong program are an inadequate means of gauging the effectiveness of a parallel programming system for speeding up real programs, because the

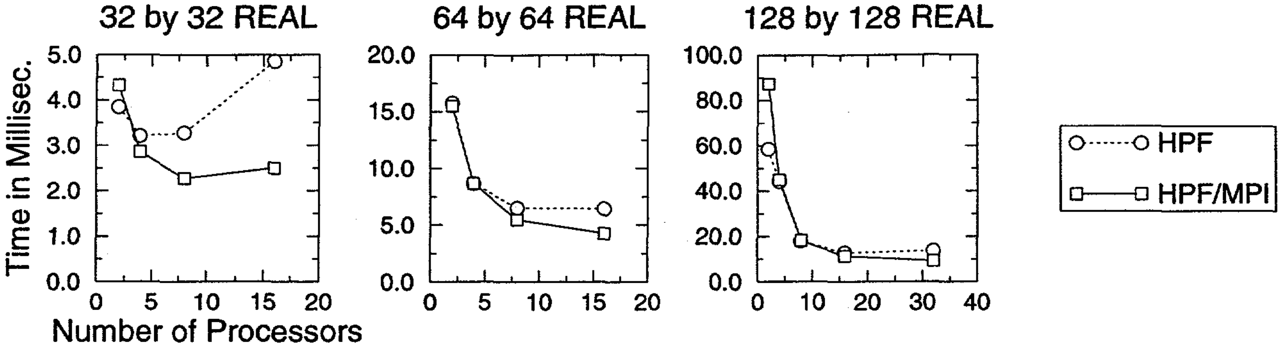

Figure 6. Time to perform 2D FFT on a single input dataset, as a function of the number of processors. Varying dataset sizes are shown.

dynamic computation and communication behavior of real programs is often different from that of microbenchmarks. Therefore, we have measured the performance of HPF/MPI using a number of application kernel benchmarks, such as pipeline codes and a multiblock code. Here we compare the performance of a pure HPF (data-parallel) and an HPF/MPI (mixed task/data-parallel) version of a two-dimensional fast Fourier transform (2D FFT) kernel.

The structure of the HPF/MPI version of 2D FFT is a pipeline containing two tasks of equal size, with one performing a (sequential) 1D FFT on each row of a matrix, then passing the matrix to a second task that performs a 1D FFT on each column. Therefore the matrix is distributed (BLOCK, *) in the first task, and (*, BLOCK) in the second, and a worst-case redistribution between tasks is required. (The structure is quite similar to that of the producer-consumer example in Figure 2, with routine produce_data performing row-wise FFTs, and consume_data column-wise ones.) In the pure HPF version, there is just one matrix which is distributed (BLOCK, *) across all processors and transposed between the two phases of 1D FFTs.

Figure 6 shows the time required by the two versions of the program to perform a 2D FFT on a single $N \times N$ matrix, for varying values of N and P. The time represents an average per dataset when a large number of datasets are processed in a single run; hence taking the reciprocals of these times yields the throughput. The performance of the HPF/MPI version is generally better. In particular, for a fixed matrix size, HPF/MPI provides an increasing improvement in speedup as P increases; for fixed P, the relative improvement in speedup of the HPF/MPI version decreases as N increases.

The performance difference between the pure HPF and HPF/MPI versions is due to higher communication overhead in the HPF version. During the matrix transpose stage of the HPF program, a message of length N^2/P^2 is exchanged between each pair of processors, so each processor sends and receives $P-1$ messages. In contrast, each processor of the HPF/MPI version must send or receive $P/2$ messages of length $4N^2/P^2$. For smaller N or larger P, message startup costs dominate total communication time, causing the HPF version with its larger number of messages to run more slowly.

On the largest matrix size plotted (128 x 128), HPF/MPI provides an improvement of up to 30% over pure HPF. While these results are promising, we believe they could be improved significantly if the overheads we have identified were reduced through further performance tuning. Another approach is to incorporate additional MPI features that let library users tune communication performance. We discuss some of these features in the next section.

4. Extending the HPF/MPI Subset

The subset MPI binding presented above includes only a small portion of the functionality of the MPI standard—just non-blocking, standard mode point-to-point communications, persistent operations, and a few simple inquiry functions such as MPI_Comm_rank. Clearly HPF/MPI's utility to programmers would be enhanced by the addition of other MPI functionality. Here we briefly consider techniques for extending the prototype design of Section 2 to incorporate features that we feel are most likely to ease development or improve performance of typical task/data-parallel applications.

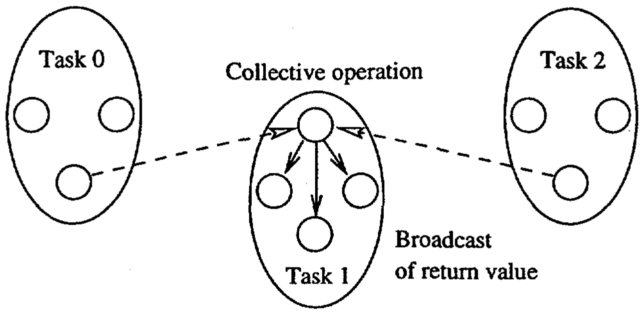

Figure 7. A collective operation involving three tasks and no distributed arguments, with task 1 the root. The return value is received only by the root task.

4.1. Collective operations

Unlike point-to-point operations, in which there is precisely one sender and one receiver, collective operations permit groups of arbitrary size to communicate using a single operation. In addition, collective operations encapsulate patterns of communication and cooperative computation such as broadcast and reduction that occur frequently in parallel applications—including HPF/MPI programs.

The complexity of performing a collective operation in HPF/MPI depends critically upon whether any of its arguments are distributed. First, we note that because barriers do not involve any transfer of user data, an HPF/MPI version may be obtained trivially through a call to the sequential version of `MPI_Barrier` by all processors that are members of tasks participating in the barrier.

The next case to consider is that in which the collective call transfers data, but none of its arguments are distributed arrays. Then one may rely on the following simple technique, illustrated by Figure 7:

1. One distinguished processor from each participating task joins in a call to the sequential version of the operation, passing its local copy of the arguments.

2. Within each participating task that is to receive the return value from the operation, the distinguished processor broadcasts the return value to all of the task's other processors. For example, for `MPI_Reduce` there is a broadcast within just the root task, while for `MPI_Allreduce` there is a broadcast within all participating tasks.

This approach is efficient for most cases and scales well.

The last case occurs when any of a collective call's arguments are distributed arrays. Such calls could be implemented as a composition of point-to-point calls using standard techniques such as combining trees, with HPF/MPI calls replacing sequential ones, where needed, to handle distributed arguments. But this simple approach misses many opportunities for optimization, because the cost of transferring a distributed array between two tasks varies greatly depending on the data distributions within each task.

As a simple illustration of the problem, suppose that Figure 7 instead depicts a single call to `MPI_Reduce` that performs a pointwise vector addition of three distributed vectors V_0, V_1, and V_2, with V_i owned by task i. Suppose further that V_0 and V_2 share the same distribution, V_1's is different, and it is expensive to convert between the two distributions. A naive implementation based on standard combining tree techniques might transfer V_0 and V_2 to task 1, so that task 1 must participate in two expensive redistributions. In many cases it will be more efficient to:

1. Transmit V_2 to task 0 (a best-case transfer involving no redistribution).

2. Compute the sum of V_0 and V_2 within task 0.

3. Transmit the partial sum to task 1, which computes the final sum.

This approach requires just one expensive redistribution.

Much more complex examples may arise in practice, as the number of ways of performing the operation grows exponentially with the number of participating tasks. To be useful, a general algorithm for selecting an efficient mapping and ordering of processing steps for a collective operation must not consume an inordinate amount of processing time or perform a large amount of communication. Development of such an algorithm appears to us to be a fundamentally hard problem.

4.2. Non-blocking communications

MPI provides many facilities for optimizing point-to-point communications. As many task/data-parallel applications depend heavily on the performance of inter-task array transfers, it is worthwhile to consider techniques for incorporating analogs of these facilities into HPF/MPI. We have already discussed the implementation of an HPF/MPI version of one such facility, namely persistent operations. We now examine non-blocking communications, which allow a sender or receiver to continue processing after a send or receive operation has been *posted*, or initiated. This feature provides two major benefits:

1. It makes possible the overlap of computation and *communication.*
2. It makes it easier for a receiver to specify a receive buffer in advance of the arrival of the message, which reduces buffer copying in some instances.

For the purposes of this discussion, we will assume that the non-blocking operations of the underlying sequential MPI implementation can provide these benefits; in practice, not all can. Given this assumption, for transfers of large arrays a non-blocking variant of the design presented in Section 2 can also provide these benefits if the data transmission step is implemented using non-blocking sequential MPI calls.

At first sight, extension of the design to provide non-blocking operations appears problematic, because there is synchronization between sending and receiving tasks during descriptor exchange. Modifying this step to use the non-blocking operations of the underlying sequential MPI library removes this synchronization, but exposes a more fundamental problem: each side can only compute a communication schedule (Step 4) after it has received the other's descriptor. Similarly, a receiver can only perform the unpacking of Step 6 after the data to be unpacked has arrived in Step 5.

In general, what is needed to permit maximum overlap between HPF/MPI library processing and application processing is some form of *message-driven execution*: the ability for some computation specified by HPF/MPI to occur upon arrival of certain messages [10]. When a message with an array descriptor arrives (Step 3), communication schedule computation should begin (Step 4), and when a data message arrives at a receiver (Step 5), it should be unpacked (Step 6). Unfortunately, within the current MPI standard the only means by which this can occur is if the application itself polls for message arrival (e.g. using `MPI_Wait` or `MPI_Test`), which is cumbersome for the programmer. Proposed support for message-driven execution in MPI-2 might alleviate this problem.

Finally, the provision of non-blocking receive operations in HPF/MPI may increase the library's buffer space requirements. Depending on the mechanism for message-driven execution, on each processor there may need to be one transfer buffer per remote processor from which data messages are to be received. This is because data messages could arrive at any time and initiate their own upacking into the destination array; hence, each message must be stored in a separate buffer to prevent corruption of one message's data by another. We address this difficulty below.

4.3. Control over system buffering

The design appearing in Section 2 provided just standard mode communications, in which the user leaves decisions about buffering and synchronization between sender and receiver up to the MPI implementation. The MPI standard includes other sending modes that provide more control over system policies, allowing the user to reduce buffer copy overhead or guarantee sufficient buffer space.

HPF/MPI can provide similar control over its resource management policies. Here we consider *buffered* mode, in which the user supplies the library with memory for buffering outgoing messages. This permits the library to complete send operations without blocking, using buffer space as necessary. The amount of space required for a message of a given size may be determined using the routine `MPI_Pack_size`. Our design for HPF/MPI requires a transfer buffer for packing messages; the underlying sequential version of MPI must also be supplied with a buffer if messages are sent using this mode. Therefore, one scheme for incorporating buffered mode sends into HPF/MPI works as follows:

- The HPF/MPI version of `MPI_Pack_size` returns a size twice that returned by the underlying sequential MPI.
- When the user supplies a buffer to the HPF/MPI library by calling `MPI_Buffer_attach`, half is used by HPF/MPI for packing messages, and the other half is supplied to the underlying sequential MPI.

To meet increased buffering requirements resulting from non-blocking receive operations, HPF/MPI could also use part of any user-supplied buffer space for transfer buffers for incoming data messages.

5. Conclusions

By utilizing a mixture of both task and data parallelism in parallel applications, one may extend the range of problems that can be solved efficiently beyond what is possible with pure data-parallel programming languages alone. We have proposed an approach for introducing task parallelism into data-parallel languages such as High Performance Fortran that makes use of a coordination library for coupling data-parallel tasks. In our case, the coordination library is a subset binding of the Message Passing Interface.

To our knowledge, this coordination library-based approach for constructing mixed task/data-parallel programs is unique. However, many other techniques have been used to introduce task parallelism into data-parallel languages. These other techniques fall into two major categories: compiler-based approaches and language-based approaches. Approaches based on compilers rely on sophisticated source code analyses and programmer-supplied directives to extract implicit task parallelism from programs [6]. In language-based approaches, language extensions permit programmers to explicitly specify the division of a computation into tasks, the mapping of tasks to processors, and communication between tasks [2]. Further comparison with other approaches appears in [3].

We have presented a design for the subset binding of MPI. Our evaluation of the performance of a prototype HPF/MPI library is encouraging: compared to a pure data-parallel HPF code for the 2D FFT, a task-parallel HPF/MPI version achieves superior performance under many parameters of execution which are of interest. However, a detailed analysis of the behavior of our library during execution of a communication-intensive microbenchmark reveals that its performance would benefit from a tighter binding with the run-time system of the HPF compiler used in our experiments, and from algorithmic extensions that would permit the library to exploit direct scatter-gather capabilities of the underlying sequential MPI substrate. An alternative is to incorporate additional MPI performance-tuning features into the library; we have suggested design techniques for several of these.

There are many promising directions for future work. Two have already been discussed: modifications to the existing prototype library to enhance performance, and extension of the current subset binding with additional MPI features that ease application development (such as collective operations) and application tuning (such as non-blocking communications). In addition, to evaluate more thoroughly the value of our techniques, we wish to construct more ambitious task/data-parallel applications than the kernels we have written up to this point. Finally, HPF/MPI provides just an explicit message-passing mechanism for inter-task interaction, yet there are many other useful mechanisms, such as single-sided operations (message-driven execution) and client-server protocols. We wish to investigate the issues involved in extending our library to incorporate some of these other mechanisms.

References

[1] 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.

[2] 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, 1994.

[3] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

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

[6] 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.

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

[8] S. Ramaswamy and P. 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., Feb. 1995.

[9] The Portland Group, Inc. *pghpf Reference Manual*. 9150 SW Pioneer Ct., Suite H, Wilsonville, Oregon 97070.

[10] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.