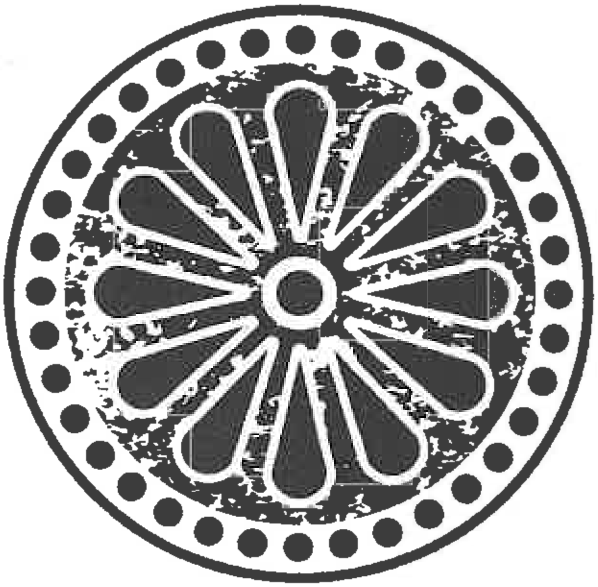
HiPC

C61

Proceedings



3RD INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING

December 19 - 22, 1996
Trivandrum, India

In cooperation with
IEEE Computer Society
IEEE Computer Society Technical Committee on Parallel Processing
acm ACM Sigarch

IEEE COMPUTER SOCIETY
50 YEARS OF SERVICE • 1946 - 1996

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Communicating Data-Parallel Tasks: An MPI Library for HPF *

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

Rakesh Krishnaiyer
Dept. of Electrical Engg. & Computer Sc.
Syracuse University
Syracuse, NY 13244
rakesh@cat.syr.edu

Alok Choudhary
ECE Dept., Technological Institute
Northwestern Univ., 2145 Sheridan Road
Evanston, Illinois 60208-3118
choudhar@cat.syr.edu

Abstract

High Performance Fortran (HPF) has emerged as a standard dialect of Fortran for data-parallel computing. However, HPF does not support task parallelism or heterogeneous computing adequately. This paper presents a summary of our work on a library-based approach to support task parallelism, using MPI as a coordination layer for HPF. This library enables a wide variety of applications, such as multidisciplinary simulations and pipeline computations, to take advantage of combined task and data parallelism. An HPF binding for MPI raises several interface and communication issues. We discuss these issues and describe our implementation of an HPF/MPI library that operates with a commercial HPF compiler. We also evaluate the performance of our library using a synthetic communication benchmark and a multiblock application.

1. Introduction

High Performance Fortran (HPF) provides a portable, high-level expression for data parallel algorithms [5]. An HPF computation has a single threaded control structure, global name space, and loosely synchronous parallel execution. Many problems that need high performance implementations are amenable to data-parallel solutions.

However, HPF does not address task parallelism or heterogeneous computing adequately. There are many applications which are not easily expressed using HPF alone [4, 2]. Examples of such applications include: multidisciplinary applications where different modules represent different scientific disciplines and may be executed on different parallel machines, applications involving irregularly structured data, and many image processing applications which are best structured as a pipeline of data parallel tasks. These applications must exploit both task and data parallelism for efficient execution on parallel machines or in a heterogeneous environment. An integrated task/data-parallel framework, where each task is a data-parallel computation, can provide improved modularity and scalability.

In this paper, we describe our design and implementation of a library-based approach to provide integration of task- and data-parallelism. Programmers call functions defined in the library for communication and synchronization between tasks. This can be contrasted with a language-based approach, where one uses explicit language constructs. Developing language extensions involves defining new syntax and semantics, enhancing the compiler's parsing and analysis phases to handle the new language constructs, and building runtime system support for them. In contrast, a library-based approach is simpler, in that it requires only that an appropriate applications programming inter-

*This work was supported in part by NSF grants CCR-9357840 and CCR-9509143.

0-8186-7557-8/96 $5.00 © 1996 IEEE

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE

face (API) be defined and implemented. The tradeoff is that the applications programmer needs to deal with the synchronization and communication details, which are handled automatically by the compiler when language extensions are used.

We use the widely accepted message passing standard MPI [3] as a coordination layer for multiple HPF tasks. Unlike the use of MPI for sequential languages, here each MPI process is in fact an HPF task executing on several processors. Hence the library provides an HPF binding for MPI. Note that MPI is used as an interface definition here, and not (necessarily) as an implementation tool.

Some important practical benefits of the HPF/MPI approach are given below:

- The library enables one to write a wide variety of SPMD style task parallel computations. Examples of such applications are multiblock codes and pipeline computations.

- The library provides a portable mechanism for transferring data between HPF computations and programs which make use of external resources for storage, data visualization, etc.

- It enables composition of new applications from existing, independent HPF programs in a manner analogous to UNIX pipes.

The next section discusses issues which arise when developing a library to support communicating data-parallel tasks. Section 3 illustrates the usage of the HPF/MPI library with an example program. The details of the library design and implementation are discussed in Sections 4 and 5. Experimental results showing the overheads of the library and other application performance results are presented in Section 6. Finally, Sections 7 and 8 review related work and present our conclusions.

2. Issues involved in data transfer

Efficient data transfer between data parallel tasks is a nontrivial problem. Sending and receiving tasks may execute on different numbers of processors and use different data distributions for communicated data structures. Tasks may execute on different computers connected by various types of networks such as Ethernet or ATM. The data to be transferred may be fully distributed, using block or cyclic distributions in one or more dimensions, or may be replicated. Finally tasks may perform a series of transfers using the same data distributions, in which case, it is useful to pre-compute and reuse communication schedules.

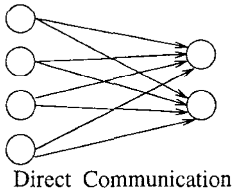


Figure 1. An efficient strategy for data transfer that relies on direct communication between senders and receivers. In this example, there are 4 processors on the sender side and 2 on the receiver side.

Different strategies can be adopted to perform the data transfer between a pair of tasks. The important factors to be taken into account include total communicated data volume, number of messages, the distribution on each side and the amount of buffering required in each processor.

An efficient strategy is to first exchange data distribution information and then perform the data transfer between the senders and receivers directly. Note that, in general, each processor on the sending side needs to communicate with a subset of processors on the receiving side and vice versa. Algorithms developed for array redistribution can be used to compute an efficient communication schedule. This strategy is illustrated in Figure 1.

3. HPF/MPI library

The basic execution model is one in which a computation consists of a collection of tasks. Each task is an HPF program executing on one or more processors. A task can also be considered as a logical MPI process. Tasks communicate and synchronize with each other using standard MPI functions for performing point-to-point or collective communication operations. The communicated data might have different HPF distributions in the participating tasks. It is the responsibility of the library to perform the data transfer between the processors involved, conforming to these distributions.

We illustrate the use of the HPF/MPI library using a simple pipelined 2D-FFT code. In this example, there is a series of 2 dimensional arrays (or images) flowing in a pipeline. For each array, we perform 1D FFTs along the columns followed by 1D FFTs along the rows. This pipeline computation can be designed to be executed as two tasks as illustrated in Figure 2. The first task performs the column FFTs; the modified array is then communicated to the second task which executes the row FFTs on the intermediate results. An HPF/MPI

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE

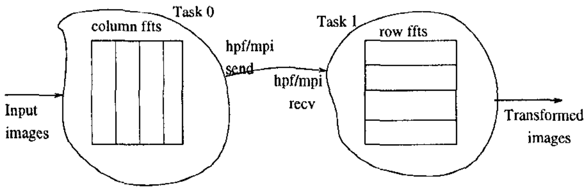


Figure 2. 2D-FFT pipeline structured as 2 tasks.

implementation of this computation is shown in Figure 3. The array has a column-wise distribution in task 0, and row-wise distribution in task 1 so that the 1D FFTs do not involve any communication.

The function MPI_COMM_SIZE returns the total number of tasks (in this case, 2) and MPI_COMM_RANK provides the task ID of the current task among all the tasks. The functions MPI_SEND and MPI_RECV are used for communication of the array between the two tasks. The programmer needs to specify, in an implementation dependent manner, the number of processors executing each task (for example, 2 processors executing task 0 and 3 processors executing task 1).

4. Details of the library

We use the direct communication strategy to perform the data transfer between the sending and receiving processors in a point-to-point HPF/MPI communication operation. The steps involved in a typical operation are as follows:

1. Each processor that belongs to a sender or receiver task determines the distribution of the communicated array. This distribution information is exchanged between the senders and the receivers.
2. Using the FALLS algorithm [7], each processor computes a set of point-to-point communication operations to be performed.
3. Each processor performs the actual set of communications computed in the previous step.

This scheme has the following benefits:

- Due to optimality of FALLS, only minimum data is transmitted between processors.
- It minimizes the total number of messages exchanged, since each sender communicates only with those receivers which require data from it, and vice-versa.

```
    program two_dim_fft
    include 'mpihpf.h'
    parameter (N=256, NITER=100)
    complex a(N,N), b(N,N)
!HPF$ processors pr(Number_Of_Processors())
!HPF$ distribute a(*,BLOCK), b(BLOCK,*) onto pr

    call MPI_Init(ierr)
    call MPI_Comm_size(MPI_COMM_WORLD,
   $                             nprocs, ierr)

!Determine which task am I: task 0 or task 1
    call MPI_Comm_rank(MPI_COMM_WORLD,
   $                              myid, ierr)

    do k = 1, NITER
      if (myid .eq. 0) then   ! column task
        forall(i=1:N, j=1:N) a(i,j)=(1.0,0.0)
!Perform the column ffts on array a
        call colfft(N,a)
!Send the intermediate result to task 1
        call MPI_Send(a,N*N,MPI_COMPLEX,1,99,
   $                    MPI_COMM_WORLD,ierr)
      else   ! I am task 1: row task
!Receive from task 0 onto array b
        call MPI_Recv(b,N*N,MPI_COMPLEX,0,99,
   $                MPI_COMM_WORLD, status,ierr)
        call rowfft(N,b) ! Perform the rowffts
        call write_output(b) ! Save the result
      endif
    end do
```

Figure 3. HPF/MPI Implementation of 2-D FFT

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE

- It keeps buffering requirements low, since the send or receive buffer required on a processor need only be as large as the largest array portion sent or received by that processor.

4.1. FALLS algorithm

The communication of a distributed data structure from one task to another can be considered as a redistribution of data from one processor subset to another. The FALLS (FAmiLy of Line Segments) algorithm uses an efficient representation of data distribution and uses novel techniques to extract a minimal sequence of communication operations to be performed to achieve this redistribution. It scales linearly with the number of dimensions and processors and handles all HPF data distributions. The communication pattern generated can be modeled as many-to-many communication between the sending and receiving processors. More details can be obtained from [7].

4.2. Optimizations

Many applications involve a series of data transfers involving the same redistribution. For example, a 2D-FFT pipeline involves communicating images with the same distribution repeatedly from one task to another. In such cases, the communication schedule generated by the FALLS algorithm can be pre-computed. This amortizes the cost of distribution information exchange and schedule generation over a number of data transfers.

We make use of the MPI *persistent requests* to provide such hints to the HPF/MPI library. An HPF program can define a persistent request using `MPI_SEND_INIT` or `MPI_RECV_INIT`. Such a function call causes the library to compute the communication schedule for such a redistribution and cache the results. The actual communication can then be performed multiple times by calling `MPI_START`.

5. Implementation

We have implemented a prototype HPF/MPI library that operates with *pghpf* (version 2.0), a commercial HPF compiler, developed by the Portland Group, Inc. We have defined an interface between our library and *pghpf* which requires only minimal modifications to *pghpf*s runtime system. This makes it easy to port our HPF/MPI library to other HPF compilers.

Most of the HPF/MPI library is written in SPMD-style C code containing explicit message-passing calls. Though HPF itself provides only a loosely-synchronous data-parallel execution model, it contains an *extrinsic* interface for performing calls to other, foreign languages which may utilize other styles of parallelism. We use the HPF extrinsic interface to gain flexibility in the implementation of our library, while retaining portability across different HPF compilation systems.

The library is structured in a modular manner using multiple levels. At the highest level, MPI calls in an application invoke functions in an HPF module within our library. Each polymorphic MPI function is represented as a Fortran 90 generic procedure. In turn, each generic procedure is implemented by a number of different HPF functions, one for each possible array rank and element type (INTEGER, REAL, COMPLEX, etc.). As an example, the blocking send operation is invoked using a function with generic name `MPI_SEND` irrespective of the datatype of the array or its rank.

These HPF library functions use HPF inquiry intrinsic routines such as `HPF_DISTRIBUTION` to determine such attributes of arrays as the extent of their dimensions, the shape of the processor grid over which they are distributed, and the form of their distribution across processors. This information is then passed on to a lower level of the library, which is written in C and therefore invoked using the HPF extrinsic interface.

At this level, the FALLS algorithm is used to compute a set of point-to-point communication operations to achieve the data transfer. Data to be transmitted must be first copied onto a contiguous buffer. In our implementation, for portability, we have used MPI as the actual transport mechanism. Nevertheless, the modular design of the HPF/MPI library makes it feasible to use a communication substrate other than MPI.

When MPI is used as the communication substrate, all processes in a computation are initiated in a manner specific to the MPI implementation. As part of HPF/MPI initialization, this set of processes is partitioned into disjoint subsets using a configuration file prepared in advance by the user. As another part of initialization, each HPF task is assigned a separate communicator, which is to be used for all internal communication required by the data-parallel computation. In addition, the data transfer between each pair of tasks makes use of another communicator. This prevents interference between communication related to task-parallel HPF/MPI calls and that related to data-parallel HPF computations.

6. Experimental Results

In this section, we present results from an evaluation of our implementation of the HPF/MPI library. These experiments were performed on Argonne's IBM

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE

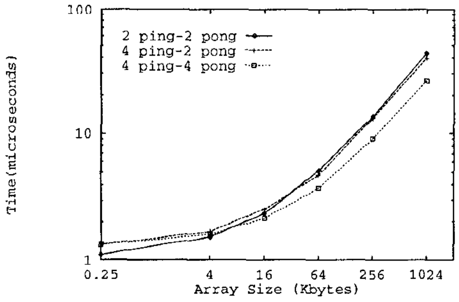


Figure 4. Point-to-point communication times between 2 tasks for different task sizes. 4 ping-2 pong denotes that 4 processors were assigned for task 0 and 2 processors for task 1.

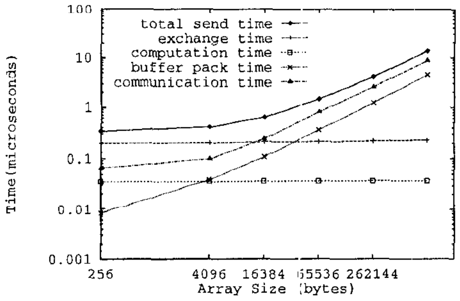


Figure 5. Time spent in different phases of a send operation.

SP system.

6.1. Synthetic ping-pong benchmark

We used the well-known ping-pong method to measure point-to-point communication times involving 2 tasks. Task 0 executes a send operation (MPI_SEND) to task 1, which executes a corresponding receive (MPI_RECV). Then task 1 immediately sends the same message back to task 0. The test program was executed a large number of times, and the mean one-way transfer time was used for interpreting the results. Each array transmitted using MPI_SEND has an HPF distribution of (*,BLOCK), and each array received using MPI_RECV has a distribution of (BLOCK,*). This data transfer pattern requires each sending process to communicate with all receivers.

Figure 4 shows the times for point-to-point communication between two tasks for different processor assignments in tasks 0 and 1. The best intertask communication bandwidth achieved in this experiment was 12.3 Mbytes/sec. This performance is comparable to that of other communication libraries which, like HPF/MPI, incur extra overhead from extra buffer copying.

The time spent in the different phases of the data transfer were measured to identify the bottlenecks involved. The different phases involved in a send operation are:

- Time spent in exchanging the data distribution information.
- Time taken for computing the set of communication operations to be performed by each processor (using the FALLS algorithm).
- Time spent in performing buffer copying.
- Time spent in performing the actual communication.

The mean total time taken by a processor in each phase is calculated by averaging over all the processors.

The break-down of the costs involved in a send operation (for the 2 ping-2 pong configuration) are shown in Figure 5. The time for performing the distribution information exchange between tasks and schedule computation are more or less constant. For large arrays, the buffer copying is a large component of the overhead. Currently, we are investigating ways to reduce this, at least for the simple cases when only contiguous portions need to be transferred.

6.2. Multiblock Application

The multiblock code is an example of an application involving irregularly structured data which is not efficiently expressed in HPF. In this code, a complex geometry is decomposed into multiple simpler blocks. A Poisson solver is run within each block and the boundary data information is exchanged between blocks periodically.

In the HPF code, the Poisson solver is run on each block one after another using all processors, whereas in the HPF/MPI version, each task handles one block, and a variable number of processors are allocated for each task. The blocks were distributed in a (*, block) fashion for both codes.

The graph in Figure 6 shows our results for three different multiblock configurations. Each configuration represents a geometry of three square blocks, where the middle block has a smaller size than the other two

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE

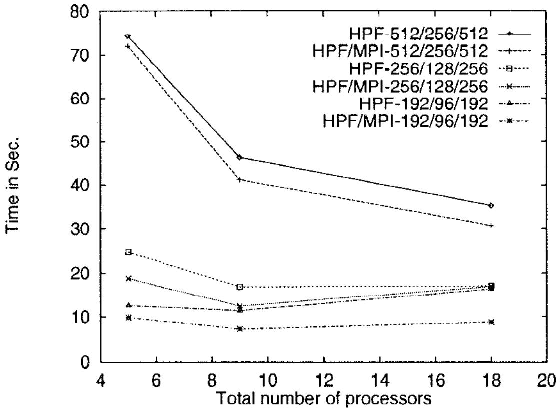


Figure 6. Execution time for HPF/MPI and HPF implementations of the multiblock code, as a function of the number of processors.

blocks. The times reported in the graph are elapsed times to convergence. In the HPF code, in each iteration, the boundary values corresponding to all blocks are exchanged, and then each of the three blocks is processed in turn. In the HPF/MPI code, at the start of each iteration, each task independently communicates with its neighboring tasks.

We can see that the HPF/MPI code performs better than the HPF code in all cases. For the 192 and 256 configurations, communication becomes the bottleneck with 18 processors and the best performance is obtained with 9 processors.

7. Related work

There has been a lot of work on language-based approaches for supporting task parallelism. Foster et. al. [1] present the language integration issues involved in mixed paradigm programming. Language projects that focus on the integration of task and data parallelism include HPF/FM[2], FX [4], and Opus [6]. HPF/FM extends HPF with channels for intertask communication. Fx creates parallel tasks that communicate by sharing arguments during task creation and termination. Opus, an extension of Fortran 90, provides a software layer on top of data parallel languages. A program executes as a system of *tasks* that interact by sharing access to a set of *shared data abstractions*.

8. Conclusions and future work

We have developed an implementation of a library based approach for supporting task parallel applications using HPF. The library opens up the possibility for HPF applications which may be heterogeneous in nature. The use of portable features of HPF in the library implementation eases the job of porting our library to other compiler systems and communication mechanisms. Empirical results show that HPF/MPI applications demonstrate better performance than equivalent pure HPF codes.

Initially, we have selected a small subset of the MPI standard to be part of the HPF/MPI library. These functions include point-to-point communication operations, enquiry functions such as `MPI_COMM_RANK` and `MPI_COMM_SIZE` and functions that provide support for persistent operations. Currently, we are investigating other MPI communication modes (such as buffered mode, asynchronous mode etc.) for inclusion into the library. Applications could also benefit from collective operations involving tasks, dynamic task management, use of MPI derived datatypes and the ability to handle array sections. Finally, we are also looking at ways to further reduce the overheads associated with the library through a tighter coupling with the *pghpf*'s runtime system.

References

[1] K. M. Chandy, I. Foster, C. Koelbel, K. Kennedy, and C.-W. Tseng. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications*, 8(2):80–98, 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, 1994.

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

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

[5] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[6] P. Mehrothra and M. Haines. An overview of the opus language and runtime system. ICASE Report 94-39, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1994.

[7] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, Feb. 1995.

Proceedings of the 1996 International Conference on High-Performance Computing (HiPC '96)
0-8186-7557-8/96 $10.00 © 1996 IEEE