A Compilation System that Integrates High Performance Fortran and Fortran M

Ian Foster,* Bhaven Avalani,† Alok Choudhary,‡ and Ming Xu*

1 Introduction

We describe a Fortran compilation system that provides integrated support for data parallelism (in the form of High Performance Fortran: HPF [5]) and task parallelism (Fortran M: FM [4]). We outline the techniques used to integrate existing HPF and FM compilers and present results from an image-processing application developed using the integrated compilation system.

While task and data parallelism are often considered opposing approaches to parallel programming, many applications can in fact benefit from both forms of parallelism. This is the case, for example, in so-called multidisciplinary applications, in which the execution of dissimiliar (and possibly data-parallel) programs must be coordinated to solve a problem. However, the development of an integrated task/data-parallel programming system introduces challenging technical issues in the areas of language design, compilation, and runtime system design.

Our compilation system allows the use of FM to coordinate concurrent HPF computations. It integrates software developed in the HPF compiler project at Syracuse and the FM compiler project at Argonne and introduces novel runtime system techniques for coupling task- and data-parallel computations. In our prototype system, data- and task-parallel procedures are clearly distinguished and are compiled separately. Simple interface routines are used to coordinate the two programming models. We are currently investigating mechanisms for a tighter integration of the two systems, with the eventual goal of producing a single language and compiler.

Some ideas developed in this work were discussed previously by Chandy et al. [2]. In other work, Cheng et al. have explored the use of AVS to coordinate CM-Fortran modules [3], while Seevers et al. use a configuration language to connect Dataparallel C computations [6]. Subhlok et al. describe a compile-time approach for exploiting task and data parallelism on iWarp [7]. Our work is distinguished by its use of an explicitly task-parallel language that supports dynamic process structures.

2 Task Parallelism

Fortran M is a small set of extensions to Fortran for task-parallel programming. Key features are its support for determinism and modularity. Programs are constructed by using single-

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. Work supported by the DOE's Office of Scientific Computing under Contract W-31-109-Eng-38.

†Department of Computer Science and NPAC, Syracuse University, Syracuse, NY 13244

‡ECE, CIS, and NPAC, Syracuse University. Work supported by NSF Young Investigator Award CCR-9357840, Intel SSD, and ARPA Contract DABT63-91-C-0028.

reader, single-writer *channels* to plug together concurrent *processes*. Processes and channels can be created and deleted, and channels can be reconnected, dynamically; nevertheless, programs can be guaranteed to be deterministic. Processes can also share variables, with access controlled by tokens to ensure deterministic execution.

FM processes can encapsulate data, computation, concurrency, and communication. This property is exploited in the HPF/FM system, in which FM processes can encapsulate data parallel HPF computations.

The Argonne FM compiler compiles FM to Fortran 77 plus calls to a thread management and message-passing library called Nexus. Nexus message-passing functions can in turn be implemented in terms of TCP/IP, shared-memory operations, MPI, or PVM, depending on the target architecture. We use a TCP/IP implementation for the experiments outlined in this paper.

3 Data Parallelism

The Syracuse HPF compiler compiles HPF programs to Fortran 77 plus calls to a set of collective communication library routines [1]. This compiler parallelizes constructs with explicit parallelism such as array constructs, forall, do-loop independent assertions, and intrinsic functions. The communication library routines are written using the Express message passing library. The resulting program can execute on any computer that supports Fortran 77 and Express.

A difficulty encountered when integrating HPF code into a task-parallel environment is that the HPF compiler assumes that the generated code is to run independently of other computations. In particular, it does not allow for multiple independent HPF computations running on the same processor, or for a HPF computation using a subset of available processors. Fortunately, we were able to address this problem by providing an FM compatibility library that implements the Express calls generated by the HPF compiler as operations on channels set up when the HPF code is invoked by an FM program. (A new FM statement, `HPFCALL`, provides this functionality.) On workstation networks, the impact on performance appears to be minor. For example, the cost of sending a message over Ethernet between two Sun SPARC-2 computers is around 10 per cent greater when using the compatibility library than when using the p4 message-passing library. We have not yet made performance comparisons on multicomputers.

4 A Case Study

Two-dimensional convolution is a simple example of a problem that can benefit from both task and data parallelism. We use it here to illustrate the integration of FM and HPF. As shown in Figure 1, 2-D convolution involves two 2-D fast Fourier transforms (FFTs), an element-wise matrix multiplication (MM), and an inverse 2-D FFT. (The figure also illustrates one of the parallel implementations explored in this paper, in which FM is used to coordinate the execution of three HPF modules.)

A data-parallel convolution algorithm can be formulated in HPF as follows. Images of size 512×512 are processed one at a time, with data distribution statements used to indicate how data parallelism is to be exploited within each FFT. The 2-D FFT uses a `REDISTRIBUTE` statement to transpose the input array so that 1-D FFTs can be performed in each dimension without communication. Each FFT operates on all processors.

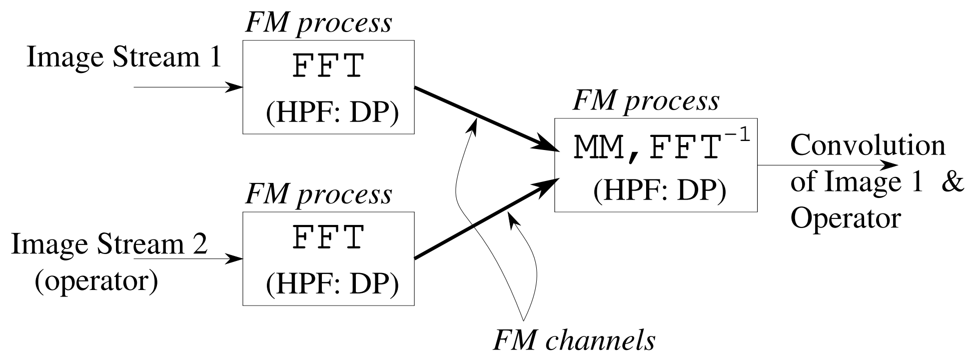

Figure 1: Convolution Algorithm Structure

```
C       HPF Implementation of Data-Parallel Algorithm
        program data_parallel
        complex A(512,512), B(512,512), C(512,512)
!HPF$   processors p(32)
!HPF$   align B, C with A
!HPF$   distribute A(*,BLOCK)
        do i = 1,nimages          ! Repeat for each image:
          call read(A,B)          ! Read input arrays from file
          call fft(A)             ! Apply FFT to first input array
          call fft(B)             ! Apply FFT to second input array
          C = A*B                 ! Element-wise matrix multiply
          forall(i=1:512, j=1:512) C(i,j) = CONJG(C(i,j))
          call fft(C)             ! Apply inverse FFT to output
          call write(C)           ! Write output to file
        enddo
```

A mixed data/task parallel algorithm can use disjoint processor subsets for each FFT, with data flowing between components in a pipeline fashion. This can improve overall performance by reducing the number of messages and by permitting computation and communication to be overlapped. The mixed algorithm uses the following FM code to implement the structure illustrated in Figure 1. This program allocates 8 processors to each FFT module. The `HPFCALL` statements create the boxes in the figure, and the channels the connections between the boxes. For simplicity, input and output are not considered. The `SUBMACHINE` annotations control resource allocation.

```
C       FM Main Program for Mixed Algorithm
        program mixed_parallel
        processors pr(24)
        outport (complex x(512,512/8)) outs1(8), outs2(8)
        inport (complex x(512,512/8)) ins1(8), ins2(8)
        channel(in=ins1(:), out=outs1(:))
        channel(in=ins2(:), out=outs2(:))
        processes
          hpfcall do_fft(nimages, outs1) submachine(1:8)
          hpfcall do_fft(nimages, outs2) submachine(9:16)
          hpfcall do_ifft(ins1, ins2) submachine(17:24)
        endprocesses
```

The program is completed by HPF wrapper code such as the following. This repeatedly initializes data, invokes the HPF FFT routine, and forwards output data. The `send` statement represents a data-parallel send: the communication of an entire distributed array (`A`) on an array of ports.

```
C       HPF Wrapper for FFT Module
        subroutine do_fft(nimages, outs)
        outport (complex x(512,512)) outs
        complex A(512,512)
        do i = 1,nimages
          call initialize(A)
          call fft(A)
          send(outs) A
        enddo
```

The FM main program and the HPF FFT routine can be compiled by using the FM and HPF compilers, respectively, to produce a Fortran 77 `main` and `f77fft` subroutine. As the HPF compiler does not currently recognize the FM `outport` or `send` constructs, we hand-compile the interface routine `dofft` to the following FM code. This receives input data, invokes the code generated by the HPF compiler (`f77fft`), and forwards output data.

```
C       FM Implementation of FFT Wrapper
        process fft(nimages,out)
        outport (complex x(512,512/8)) out
        complex Alocal(512,512/8)
        do i = 1,nimages
          call initialize(Alocal)
          call f77fft(Alocal)
          send(out) Alocal
        enddo
```

This code is then compiled with the FM compiler and linked with `main`, `f77fft`, and various FM libraries to produce an executable program. We are currently investigating the automatic generation and compilation of interface routines.

In Figure 2, we give preliminary performance results for the data-parallel and mixed codes. These results were obtained on an IBM SP1 using TCP/IP for interprocessor communication. The results can be explained as follows. The data-parallel code communicates less data than the mixed code, but sends more messages. The data parallel code's lower communication volume makes it faster for larger N and smaller P, while the mixed code is faster when the cost of sending many messages dominates, that is, for smaller N and larger P.

5 Summary and Further Work

The full paper will provide a more detailed description of the techniques used to develop the integrated task/data-parallel compilation system, present additional performance studies, and use a performance model to account for the performance differences observed in the two algorithms. In future work, we will investigate tighter integration of task- and data-parallel programming, in terms of both language design and implementation techniques.

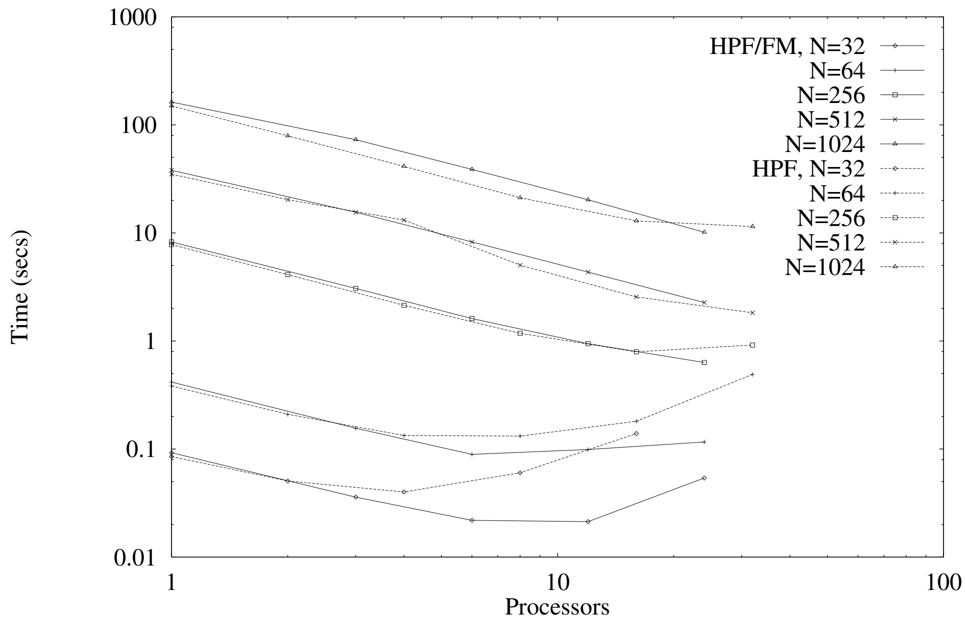

Figure 2: Convolution Code Performance on IBM SP1

References

[1] Z. Bozkus, Alok Choudhary, Geoffrey C. Fox, T. Haupt, and S. Ranka, Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. *Proc. Supercomputing '93*, ACM, 1993.

[2] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, Integrated support for task and data parallelism, Preprint MCS-P384-0993, Argonne National Laboratory, and Technical Report, CRPC, Rice University, Houston Texas. 1993.

[3] G. Cheng, G. Fox, and K. Mills, Integrating multiple programming paradigms on Connection Machine CM5 in a dataflow-based software environment, Technical Report, NPAC, Syracuse University, Syracuse, N.Y., 1993.

[4] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992, Argonne National Laboratory, June 1992.

[5] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.

[6] B. Seevers, M. Quinn, and P. Hatcher, A parallel programming environment supporting multiple data-parallel modules, *Intl J. Parallel Programming*, 21(5), 363–386, 1992.

[7] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross, Exploiting task and data parallelism on a multicomputer, *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, 1993.