Implementing Pipelined Computation and Communication in an HPF Compiler

Thomas Brandes[1]*
Frédéric Desprez[2]**

[1] GMD, SCAI
Schloss Birlinghoven, P.O. Box 1319, 53754 St. Augustin, Germany
[2] LIP, ENS Lyon, CNRS URA 1398, INRIA Rhône-Alpes
46, Allée d'Italie, 69364 LYON cedex 07, France

Abstract. Many scientific applications can benefit from pipelining computation and communication. Our aim is to provide compiler and runtime support for High Performance Fortran applications that could benefit from these techniques. This paper describes the integration of a library for pipelined computations in the runtime system. Results on some application kernels are given.

1 Introduction

With the introduction of High Performance Fortran (HPF) [KLS+94], it is possible to use the data parallel programming paradigm in a very convenient way for scientific applications. With current compilation technology, these programs will execute phases of computations and communications on differents sets of data and no overlap exists between communications and computations. Moreover, communication phases are synchronous, i.e. each processor executes these phases at the same time and waits until the last processor completes his communication phase. An important task of the HPF compiler is to detect the potential of overlapping computation and communication and to take efficient use of it.

Overlapping is not always possible because of the dependences within the code. In this case the computation might be broken into smaller pieces that can be executed in a pipelined fashion. This is also called macro-pipelining [Kin88, Tse93]. Usually, the resulting code of macro-pipelining is very complicated. But we will show that it is possible to use runtime system functions that do this splitting at runtime. This does not only decrease the complexity of the HPF compiler, but also allows the optimization of overlapping computation and communication at runtime. This can be done by making some runtime measurements that determines the best size of granularity.

Though most of the techniques are already known, this paper focus on the efficient use and the integration in an existing HPF compilation system. The runtime functions are a new version of the LOCCS library (Low Overhead Communication and Computation Subroutines) that has been first presented in [DT92].

* This works has been supported by the Esprit-6643 project PPPE (Portable Parallel Programming Environment)
** This work has been supported by the INRIA Rhône-Alpes and the PRC-GDR PRS

2 Pipelined Computations

In pipelined computations, a processor cannot begin execution until it receives results computed by its predecessor. Though this kind of pipelined execution for its own is still a sequential execution, there are two possibilities to extract partial parallelism by overlapping computations. If one pipelined computation follows another one, all processers become busy when the pipeline is filled. The other possibility is to break the computation and to send partial results. By this way the processors may overlap their computations as shown in Figure 1.

The ADI algorithm in Figure 2 is a typical example that benefits from pipelining computation. Assume that the columns are distributed in a block fashion among the available processors. The first loop nest contains no dependence and can be executed in parallel. In the second loop nest, every processor computes the results in row order, sending the last value to the next processor as soon as it is ready. This strategy produces a pipelined effect.

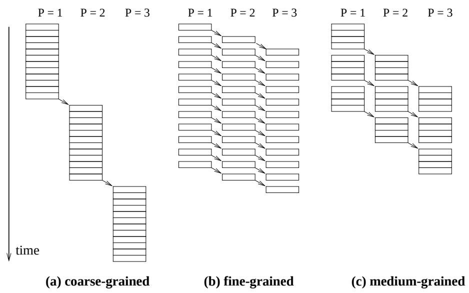

Fig. 1. Breaking up a pipelined computation.

```
       PARAMETER (N=...)
       REAL, DIMENSION (N,N) :: A, B
!HPF$  DISTRIBUTE (*, BLOCK) :: A, B
       DO I= 2, N
         DO J= 1, N
           A(I,J) = A(I,J) − A(I−1,J)*B(I,J)
         END DO
       END DO
       DO J = 2, N
         DO I = 1, N
           A(I,J) = A(I,J) − A(I,J−1)*B(I,J)
         END DO
       END DO
```

Fig. 2. ADI algorithm in HPF.

Usually, this method is not very efficient due to the large communication startup time on MIMD message-passing machines. Therefore, a variant of the method is chosen where each processor computes a few rows before communicating the results (tiling).

3 Implementation within ADAPTOR

ADAPTOR (Automatic Data Parallelism Translator) is a public domain compilation system developed at GMD for compiling data parallel HPF programs to equivalent message passing programs [BZ94].

Instead of generating directly complex pipelining code for loop nests, ADAPTOR provides a driver routine that is implemented within the runtime system DALIB.

This routine is called with the sections belonging to the iteration space and with the corresponding data dependences. The first parameter is the name of the local subroutine that contains the code for one tile. The compiler has to find cross-processor loops and must generate the call to the routine as well as the code for the local subroutine.

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N) :: A, B
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
      ...
      DO I = 2, N
        DO J = 1, N
          A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
        END DO
      END DO
      CALL DALIB_LOCCS_SHIFT (BLOCK, 2, 0,
          A(:,2:N), [0,1], B(:,2:N), [0,0])
```

```
      EXTRINSIC (HPF_LOCAL) SUBROUTINE BLOCK (A, B)
      REAL A(:,:), B(:,:)
!HPF$ DISTRIBUTE *(*,BLOCK) :: A, B
      DO J=lbound(A,2),ubound(A,2)
        DO I=lbound(A,1),ubound(A,1)
          A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
        END DO
      END DO
      END
```

The runtime approach allows to integrate machine dependent optimizations in the runtime system. Furthermore, the driver can deal with arbitrary distributions of the arguments. Therefore we can generate code also for cases where the distribution of the arguments is unknown at compile time. The computation of the optimal size of the tiles can be computed at run-time and dynamically adjusted.

A detailed description of the interface and of the implementation is given in our report [BD96].

4 Results

In this section, we give the results of first experiments using optimized pipelined computations within ADAPTOR.

Figure 3 shows the speedups achieved by pipelining on the IBM SP 2 (AIX 3.2.5). We compare the execution time of the parallel program against the execution time of the serial program to show the real speed-ups. The pipelined execution uses in the serial dimension the block size 16.

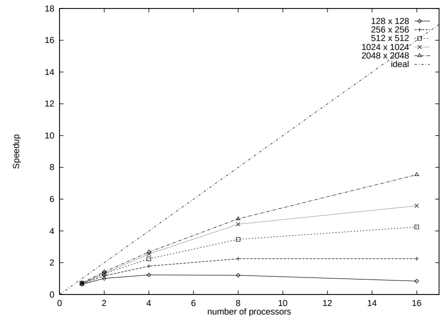

Fig. 3. Speedups for the pipelined execution of the gauss-seidel relaxation.

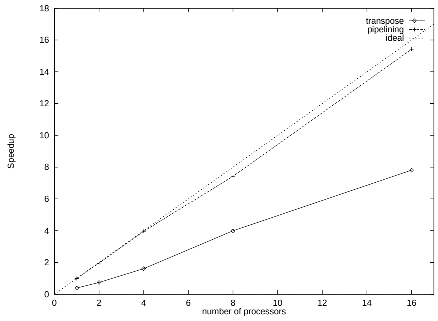

Fig. 4. Speedups for the ADI algorithm.

Now we discuss the parallelization of the ADI algorithm given on Figure 2. This algorithm contains dependences partly between the columns, partly between the rows. There are two strategies to solve this problem, one using a redistribution (transposition) of the array, one utilizing pipelined execution. As shown in Figure 4 the pipelined execution achieves nearly the optimal speed-up and needs no temporary data.

Our report describes other applications of pipelined executions and gives more detailled results [BD96].

5 Conclusion and Future Work

Our first impressive results show that there is no doubt about the usefulness of pipelining and about the efficient realization within ADAPTOR. Our experiments have shown that the pipelined computation using the overlap of computation and communication can be integrated successfully within an HPF compiler. Using these kind of optimizations in a message passing program is difficult and usually machine-dependent. By their integration in a compiler, the user can benefit from it a very convenient way. The interface of the LOCCS library is now well suited for the optimization of a compiler like ADAPTOR. An MPI version of the LOCCS is currently under development. Our future work concentrates on increasing the possibilities of pipelined executions and on the development of good algorithms and heuristis for choosing the order and the grain of the tiles.

Acknowledgements

We thank ZAM, Jülich for providing access to the Intel Paragon XP/S.

References

[BD96] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. Technical report, LIP - ENS Lyon, 1996.

[BZ94] Th. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser, April 1994.

[DT92] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. Technical Report 92-44, LIP - ENS Lyon, December 1992.

[Kin88] C. King. *Pipelined data parallel algorithms: Concept, design and modeling.* PhD thesis, Michigan State University, Department of Computer Science, 1988.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[Tse93] C. Tseng. *An optimizing Fortran D compiler for MIMD distributed-memory machines.* PhD thesis, Rice University, Houston, Texas, 1993.