The Pandore Data-Parallel Compiler and its Portable Runtime

Françoise André, Marc Le Fur, Yves Mahéo, Jean-Louis Pazat*

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, FRANCE

Abstract. This paper presents an environment for programming distributed memory computers using High Performance Fortran. Emphasis is put on compilation techniques and distributed array management. Results are shown for some well known numerical algorithms.

1 Introduction

The difficulty of programming massively parallel architectures with distributed memory is a severe impediment to the use of these parallel machines. In the past few years, we have witnessed a substantial effort on the part of researchers to define parallel programming paradigms adapted to Distributed Memory Parallel Computers (DMPCs).

Among them, the *Data Parallel* model is an interesting approach: the programmer is provided a familiar uniform logical address space and a sequential flow of control. He controls the distributed aspect of the computation by specifying the data distribution over the local memories of the processors. The compiler generates code according to the SPMD model and the links between the code execution and the data distribution is enforced by the *owner-writes* rule: each processor executes only the statements that modify the data assigned to it by the distribution. This approach constitutes the basis of several compilers [15, 17] and is also applied in the PANDORE compiler.

This paper presents the PANDORE environment and focuses on some optimizations of the compilation scheme and the run-time support. It is organized as follows: the next section presents the PANDORE environment. Section 3 explains the compilation process and the optimizations used for parallel nested loops. The array management is described in section 4 and results are discussed in section 5. Future work and extensions of our system are presented in the conclusion.

2 The Pandore Environment

The PANDORE environment is shown in figure 1. It comprises a compiler, a machine-independent run-time and execution analysis tools including a profiler and a trace generator. The HPF front-end is built upon a Fortran 90 precompiler designed at GMD [11]. The PANDORE run-time uses a generic message passing

* E-mail: pandore@irisa.fr

library called POM (Parallel Observable Machine) [8]. This library offers limited but efficient services. It allows the same program to run on a wide range of distributed memory computers.

The source language is a subset of High Performance Fortran. HPF includes some "standard" extensions for data parallelism and data distribution into the Fortran 90 language. They permit to distribute arrays among virtual processor arrays. Examples of HPF programs are shown in section 5, see [7] for a complete description of the language.

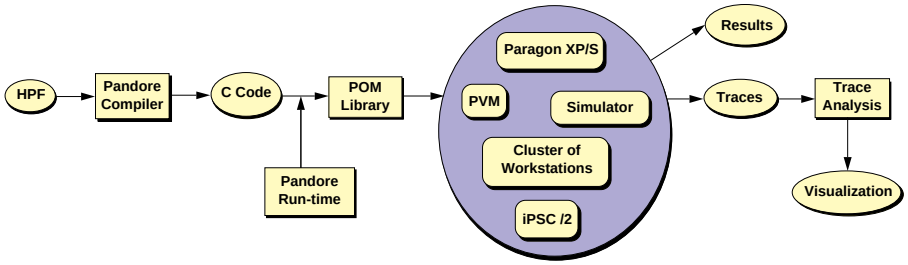


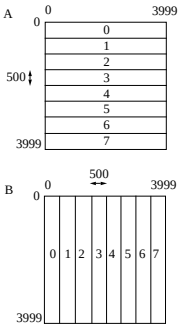
Fig. 1. The PANDORE Environment

3 The Pandore Compiler

When we designed the compiler, one of our objectives was to build a modular and extensible compiler in order to be able to integrate new techniques easily. The compiler has been written using the CAML language [2] that is a dialect of the functional language ML.

The first prototype of the PANDORE compiler relied on the well-known *run-time resolution* technique [4] that has been proved correct [3]. This technique introduces masks and potential communications at the statement level. Consequently, it suffers from strong inefficiencies. So an optimized compilation scheme handling reductions and parallel loops with one statement has been added to the current version. A suitable run-time system completes these compilation techniques and permits to obtain efficient execution for a reasonably large class of programs.

Compiling Reductions and Parallel Loops. The compilation scheme is based on the decomposition of the arrays into blocks; it performs the restriction of iteration domains and the vectorization of messages. The mapping is taken into account at run-time. A complete description can be found in [13]. In this section, we only describe our approach through the following (somewhat contrived) example where arrays A and B are partitioned into 8 blocks numbered from 0 to 7:




```
      REAL, DIMENSION(0:3999,0:3999) :: A,B
!HPF$ PROCESSORS PROCS(P)
!HPF$ DISTRIBUTE (CYCLIC(500),*) ONTO PROCS :: A
!HPF$ DISTRIBUTE (*, CYCLIC(500)) ONTO PROCS :: B

      DO I=1, 1000
         DO J=1, 2*I+1
            A(I,J-1) = B(J,I+J-2)
         END DO
      END DO
```

For this parallel nested loop, the compiler produces a SPMD code that comprises a communication part followed by a computation part.

Communication Code Generation. First, the compiler performs a symbolic analysis on the iteration domain and the array subscripts, allowing for the partitioning and the layout of the arrays, in order to construct the following system of affine constraints:

$$\begin{cases} 0 \leq kA \leq 7 & 1 \leq i \leq 1000 & u = j & 500 * kA \leq i \leq 500 * kA + 499 \\ 0 \leq kB \leq 7 & i \leq j \leq 2 * i + 1 & v = i + j - 2 & 500 * kB \leq v \leq 500 * kB + 499 \end{cases}$$

that defines the set of vectors (kA, kB, v, u, i, j) in which the array element $B(u, v)$, located in the block number kB of B, is needed to perform the writings on the the block number kA of A. The above system of constraints defines a polyhedron P which is then projected along the i, j axis; this results in a new polyhedron P_{ij} whose enumeration code can be computed by one of the methods described in [12, 9, 6]. This yields the nested loop (in pseudo-code):

for $kA = 0\ ,\ 2$
  for $kB = max(0, 2 * kA - 1)\ ,\ min(5, 3 * kA + 2)$
    for $v = max(500 * kB, 1000 * kA - 2)\ ,\ min(500 * kB + 499, 1500 * kA + 1496)$
      for $u = max(div(v + 3, 2), v - 998, -500 * kA + v - 497)\ ,$
        $min(div(2 * v + 5, 3), -500 * kA + v + 2)$

From this loop, the compiler generates the send code and the dual receive code for reference $B(j, i + j - 2)$. For the send code for instance, the compiler inserts *send* statements and appropriate masks in the loop so that the mapping of the blocks is taken into account at run-time:

for $kA = 0\ ,\ 2$
  if $myself \neq$ owner_block$(A,\ kA)$
    for $kB = max(0, 2 * kA - 1)\ ,\ min(5, 3 * kA + 2)$
      if $myself =$ owner_block$(B,\ kB)$
        for $v = max(500 * kB, 1000 * kA - 2)\ ,\ min(500 * kB + 499, 1500 * kA + 1496)$
          RLR_send $\{B(u, v)\ /\ u \geq max(div(v + 3, 2), v - 998, -500 * kA + v - 497)$
            $u \leq min(div(2 * v + 5, 3), -500 * kA + v + 2)\}$
            to owner_block$(A,\ kA)$

Although this loop contains masks, it is important to notice that these masks are evaluated at the block level and not at the iteration vector level as in the run-time resolution. Furthermore the (kA, kB)-loop do not scan the whole cartesian product $0..7 \times 0..7$ and the location of the first mask prevents from enumerating all the vectors described by the (kA, kB)-loop.

From the description of the elements $B(u, v)$ to be sent, the run-time library routine *RLR_send* performs several communication optimizations. Direct communication is performed when possible: what is transferred in this case is a memory zone that is contiguous both on the sender and the receiver side, thus eliminating any need of coding/decoding or copying between message buffers and local memories. Message aggregation is also carried out and reduces the effect of latency by grouping small messages into a large message. Finally, the redundant communications, that may occur when several references to the same distributed array appears in the right hand side, are eliminated.

Computation Code Generation. The SPMD computation code is generated as follows: the compiler analyzes the iteration domain, the subscripts and the array partitioning for the reference $A(i, j-i)$ to synthesize the set of constraints:

$$\begin{cases} 0 \leq kA \leq 7 & 1 \leq i \leq 1000 \\ kA \leq i \leq 500 * kA + 499 & i \leq j \leq 2 * i + 1 \end{cases}$$

that defines the set of vectors (kA, i, j) where iteration vector (i, j) is such that reference $A(i, j - i)$ writes in block kA of A. The enumeration code for the polyhedron associated with the previous system is then computed as in the communication code generation:

for $kA = 0\ ,\ 2$
  for $i = max(500 * kA, 1)\ ,\ min(500 * kA + 499, 1000)$
    for $j = i\ ,\ 2 * i + 1$

From this nested loop, the computation code is finally generated by inserting an adequate mask so that the owner-writes rule is ensured at run-time:

for $kA = 0\ ,\ 2$
  if $myself =$ owner_block$(A,\ kA)$
    for $i = max(500 * kA, 1)\ ,\ min(500 * kA + 499, 1000)$
      for $j = i\ ,\ 2 * i + 1$
        $A(i, j - i) := B(j, i + j - 2)$

As in the communication code, one can note that the number of tests performed by each processor is very small. First, the mask used to take into account the mapping at run-time is introduced at the block level and second, the outer kA-loop does not scan the whole interval $0..7$.

4 Management Scheme for Distributed Arrays

Representation of distributed arrays as well as accesses to elements of these arrays is a critical issue for overall performance of the produced code. In the

PANDORE environment, arrays are managed by a *software* paging system. The run-time uses the addressing scheme of standard paging systems but is not a virtual shared memory: the compiler always generates communication when distant data are needed, so we do not need to handle page faults.

The array management is based on the paging of arrays – not of memory: the multi-dimensional index space of *each* array is linearized and then broken into pages. Pages are used to store local blocks and distant data received. If data have to be shared by two processors, each processor stores a copy of the page (or a part of the page) in its local memory. Array elements are accessed through a table of pages allocated on each processor. The compilation technique ensures that accessed pages are up to date, hence the consistency between copies of array elements does not need to be handled at run-time.

Description. To access an element referred to by an index vector $(i_0, \ldots, i_{n-1})$ in the source program, a page number and an offset (PG and OF) are computed from the index vector with the linearization function $\mathcal{L}$ and the page size S: $PG = \mathcal{L}(i_0, \ldots, i_{n-1})\ div\ S$, $OF = \mathcal{L}(i_0, \ldots, i_{n-1})\ mod\ S$. For a given distributed array, the parameters we tune for paging are the page size S and the linearization function $\mathcal{L}$. Time consuming operations are avoided in the computation of the tuple (PG,OF) but also in the evaluation of the function $\mathcal{L}$ by introducing powers of two, turning integer division, modulo and multiplication into simple logical operations (shift and mask). We first choose the dimension δ in which the size of the blocks is the largest. Function $\mathcal{L}$ is the C linearization function applied to a permutation of the access vector that puts index number δ in last position. The page size S is then defined by the following (s_δ is the block size in dimension δ): if s_δ is a power of two or dimension δ is not distributed, S is the smaller power of two greater than s_δ; otherwise S is the largest power of two less than s_δ. Actually, an optimized computation of (PG,OF) is achieved by avoiding the explicit computation of the linear address $\mathcal{L}(i_0, \ldots, i_{n-1})$: we express PG and OF directly as a function of the index vector, thus, when dimension δ is not distributed, *mod* and *div* operations are removed. A more detailed description of this array management can be found in [14].

Benefits. This management scheme leads to an efficient access mechanism. Access times remain very close to access times without index conversion and may be an order of magnitude faster than a "classical" index conversion involving a modulo or an integer division. The memory overhead induced by paging does not exceed a few percents for most distributions; it is almost entirely due to the tables of pages: when a page contains elements that have no equivalent in he original sequential space, or when just a part of a distant page is accessed in a loop, only a portion a the page is actually allocated.

Apart from the performance aspects, paging distributed arrays offers several worthwhile characteristics. First, the scheme is always applicable. It is also independent of the analysis of the code: it only depends on distribution parameters, so no data re-arrangement or extra calculation is needed withing the scope of one

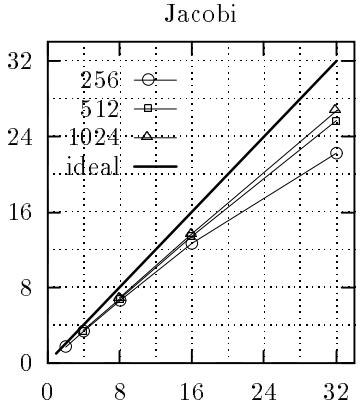
distribution even if different loop and access patterns are concerned. The scheme is uniform: as far as accesses are concerned, no difference is made between local elements and distant elements previously received. Finally, the memory contiguity is preserved in the direction of the pages: contiguous elements of the original array are still contiguous in the local representation, facilitating direct communications and exploitation of caches and vector processors.

To our knowledge, management of distributed arrays have not been studied independently of compilation techniques. Several techniques such as the overlap, temporary buffers and hash tables are presently used in existing HPF-like compilers [16, 15, 17]. Although they have not been integrated in complete prototypes, other techniques that aims at packing local data to decrease memory overhead have been proposed [10, 5]. None of these methods gather all the characteristics discussed above.

5 Experimental Results

The compilation of several well-known kernels and larger applications have been tested with the PANDORE environment. Performance results are shown here for three kernels: Cholesky factorization, Matrix-matrix product and Jacobi relaxation; the description of the parallelization of a wave propagation application can be found in [1]. The source code of the HPF subroutines are given. Apart from the distribution specification, they include minor modifications compared with the original sequential code. These modifications are to a large extent aimed at taking advantage of collective communications.

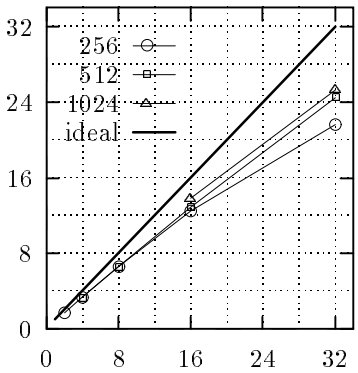
Measurements have been performed on a 32-node iPSC/2. The presented graphs show the speedup against the number processors for several input sizes (the indicated number is the value of N). Speedup is defined as the parallel time over the time of the original sequential program measured on one node. The obtained efficiencies are satisfactory, ranging from 85% to 95% on 8 processors and reaching around 80% on 32 processors for the largest data size.



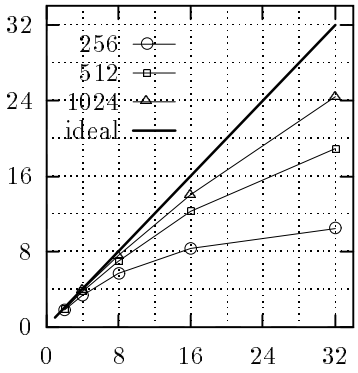

```
SUBROUTINE JACOBI (B)
  INTEGER, PARAMETER :: N = 512
  REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A, B
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO PROCS :: A, B
  INTEGER I, J

  DO J=1, N-2
     DO I=1, N-2
        A(I,J) = 0.5 * B(I,J) + 0.125 * &
        & (B(I-1,J) + B(I+1,J) + B(I,J-1)+ B(I,J+1))
     END DO
  END DO

  B(1:N-2, 1:N-2) = A(1:N-2, 1:N-2)
END SUBROUTINE JACOBI
```




```
SUBROUTINE MATPROD (A,B,C)
  INTEGER, PARAMETER :: N = 512
  REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A, B, C
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO PROCS :: A, C
!HPF$ DISTRIBUTE (*, BLOCK) ONTO PROCS :: B
  INTEGER I,J,K
  REAL (KIND=8), DIMENSION (0:N-1) :: COLJ
  C(0:N-1,0:N-1) = 0.0
  DO J=0, N-1
    COLJ = B(0:N-1,J)
    DO I=0, N-1
      DO K=0, N-1
        C(I,J) = C(I,J) + A(I,K) * COLJ(K)
      END DO
    END DO
  END DO
END SUBROUTINE MATPROD
```




```
SUBROUTINE CHOLESKY (A)
  INTEGER, PARAMETER :: N = 512
  REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE (*, CYCLIC) ONTO PROCS :: A
  INTEGER I,J,K
  REAL (KIND=8), DIMENSION (0:N-1) :: COLK
  DO K=0, N-1
    A(K,K) = SQRT(A(K,K))
    DO J=K+1, N-1
      A(J,K) = A(J,K) / A(K,K)
    END DO
    COLK(K+1:N-1) = A(K+1:N-1,K)
    DO J=K+1, N-1
      DO I=J, N-1
        A(I,J) = A(I,J) - COLK(I) * COLK(J)
      END DO
    END DO
  END DO
END SUBROUTINE CHOLESKY
```

6 Conclusion

Thanks to the above described optimization techniques, the performances obtained on a series of numerical applications are already quite satisfactory even though enhancements can be made along several axis. For example, taking the mapping of the blocks into account at compile time will allow us to suppress masks in the loops generated in the communication and the computation codes. Moreover, we plan to enlarge the subset of HPF compiled, especially by adding alignments and nested subroutines calls.

To handle very large applications, in particular applications that comprise multiple modules or necessitate intensive I/O, other techniques must be integrated to the existing environment. For this purpose, the joint study of redistribution, procedures and separate compilation is under way in the PANDORE project.

We think that these improvements will contribute to a better maturity for data parallel compilers and so automatic code generation for distributed memory parallel architectures will become a realistic means of programming these architectures for application users.

References

1. F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. Parallelization of a Wave Propagation Application using a Data Parallel Compiler. In *IPPS '95*, Santa Barbara, California, April 1995.
2. M.V. Aponte, A. Lavaille, M. Mauny, A. Suarez, and P. Weis. *The CAML Reference Manual.* Technical Report 121, INRIA, September 1990.
3. C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. *Correctness of Automated Distribution of Sequential Programs.* Research Report 665, IRISA, France, June 1992.
4. D. Calahan and K. Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, October 1988.
5. S. Chatterjee, J.R. Gilbert, F.J.E. Schreiber, and S.H. Teng. Generating Local Adresses and Communication Sets for Data-Parallel Program. In *The Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, July 1993.
6. J.F. Collard, P. Feautrier, and T. Risset. *Construction of DO Loops from Systems of Affine Constraints.* Research Report 93-15, LIP, Lyon, France, 1993.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.
8. F. Guidec and Y. Mahéo. *POM: a Virtual Parallel Machine Featuring Observation Mechanisms.* Research Report 902, IRISA, France, January 1995.
9. F. Irigoin and C. Ancourt. Scanning Polyhedra with DO Loops. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.
10. F. Irigoin, C. Ancourt, F. Coelho, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *International Workshop on Compilers for Parallel Computers*, December 1993.
11. L. Jerid, F. André, O. Chéron, J.-L. Pazat, and T. Ernst. *HPF to C-Pandore Translator.* Technical Report 2283, INRIA, France, May 1994.
12. M. Le Fur. *Parcours de polyèdre paramétré avec l'élimination de Fourier-Motzkin.* Research Report 858, IRISA, Rennes, France, September 1994.
13. M. Le Fur, J.L. Pazat, and F. André. *Static Domain Analysis for Compiling Commutative Loop Nests.* Research Report 2067, INRIA, France, October 1993.
14. Y. Mahéo and J.-L. Pazat. *Distributed Array Management for HPF Compilers.* Research Report 2156, INRIA, France, December 1993.
15. C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.
16. H. P. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, (6):1–18, 1988.
17. H. P. Zima and B. M. Chapman. *Compiling for Distributed-Memory Systems.* Research Report APCP/TR 92-17, Austrian Center for Parallel Computation, University of Vienna, November 1992.