I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 868

PARALLELIZATION OF A WAVE PROPAGATION APPLICATION USING A DATA PARALLEL COMPILER

FRANÇOISE ANDRÉ, MARC LE FUR, YVES MAHÉO,
JEAN-LOUIS PAZAT

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Parallelization of a Wave Propagation Application using a Data Parallel Compiler

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

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet PAMPA

Publication interne n ° 868 — Octobre 1994 — 14 pages

Abstract: This paper presents the parallelization process of a Wave Propagation application using the PANDORE environment. Tools are briefly described, the stress is put on the description of the parallelization by data distribution.

Key-words: Compilation, Distributed Memory Machines, Distribution, Parallelization, Applications.

(Résumé : tsvp)

*fandre@irisa.fr
**mlefur@irisa.fr
***maheo@irisa.fr
****pazat@irisa.fr

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Parallélisation d'une application de propagation d'ondes à l'aide d'un compilateur data-parallèle

Résumé : Ce rapport présente la parallélisation d'une application de propagation d'ondes effectuée à l'aide de l'environnement PANDORE. Les outils sont brièvement décrits puis l'accent est mis sur la description de la parallélisation par distribution de données.

Mots-clé : Compilation, Machines à mémoire distribuée, Distribution, Parallélisation, Applications.

1 Introduction

The difficulty to program massively parallel architectures with distributed memory is a severe impediment to the use of these parallel machines. In the past few years, the *data parallel* model has been used to define new languages such as HPF [8], tools and compilers: the programmer is provided a familiar uniform logic address space and a sequential flow of control. He controls the distributed aspect of the computation by specifying the data distribution on 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*.

To achieve good performance when following this approach, sophisticated compilation techniques and run-time systems have been studied and integrated into environments [11, 4, 2]. Other related techniques have been proposed [1, 6, 13] but have not been fully integrated yet in complete environments.

Based on the data parallel approach, the PANDORE environment allows compiling both HPF and PANDORE programs into SPMD machine independent code. A series of experiments on classical kernels have already led to satisfactory results. The next step is naturally the validation of the compiler and the run-time system on "real applications"; the Wave Propagation application presented here is one of them.

The paper is organized as follows: we briefly present the PANDORE environment and give an overview of the C-PANDORE language. The compilation schemes are then expounded and the distribution of the well-known Jacobi kernel is detailed. Finally, the different steps of the parallelization of the Wave Propagation application are described and the results of this experiment are discussed.

2 The Pandore Environment

The PANDORE environment has been designed to facilitate the programming of data distributed applications for distributed memory computers or clusters of workstations. Figure 1 shows the components of the PANDORE environment.

The source program can be written in HPF or in a dialect of C (C-PANDORE) augmented with data distribution features. In the first case, a source to source translator is used to translate a subset of HPF into C-PANDORE. This translator [14] is built upon a Fortran 90 Front-end and a C-code generator written at GMD-First with the Cocktail toolbox [9].

From the source program, the PANDORE compiler automatically generates a machine independent SPMD code according to the owner-writes rule: a processor executes only the statements that modify variables assigned to it by the distribution. Optimizations are included in the compiler that is described in section 4. For performance measurements we also provide a profiler and a post-mortem analysis tool.

The PANDORE run-time is in charge of the management of distributed arrays [17]; it uses a generic message passing library called POM (Parallel Observable Machine) [10]. This library offers limited but efficient services. It allows running the same program on a wide range of

A PANDORE program is a sequential program that calls distributed phases. The sequential part is in charge of all I/O operations and is executed on the host processor (if it exists) or on one node of the distributed computer. Each distributed phase is spread over the processors of the target machine and is executed in parallel. The specification of a distributed phase is described similarly to the definition of a procedure by the statement:

dist *d-phase* (*distributed parameter list*)
{
d-block
}

The distributed parameter list allows specifying the partitioning and the mapping of the data used in the distributed phase. The array is the only data type that may be partitioned. The means to decompose an array is to split it into blocks. The number of blocks is independent of the number of processors: both `BLOCK` and `CYCLIC(`k`)` HPF distribution features [8] are handled.

The specification of the partitioning for a d-dimensional array is given by the construct **block** $(t_1, ..., t_d)$ where t_i indicates the size of the blocks in the i^{th} dimension. For example

$B[N][N]$ **by block** $(N, N/P)$

indicates that the array B of $N \times N$ elements is decomposed into P blocks of size $N \times N/P$: the array is decomposed into P blocks of contiguous columns. The following partitioning

$A[N][N]$ **by block** $(1, N)$

indicates that the array A of $N \times N$ elements is decomposed into blocks of size $1 \times N$: the array is decomposed into N rows.

Then, the mapping of the blocks onto the architecture will be achieved by the compiler in a regular or cyclic way according to the mapping parameters (**regular** or **wrapped**). In PANDORE, we consider only one dimensional processor arrays whose size is not specified in the source code but used as a parameter by the compiler. As we allow the mapping of multidimensional decompositions, it is needed to indicate the order for the mapping of blocks: `(1,0)` states for column first, `(0,1)` states for row first. For example

double $A[N][N]$ **by block**$(1, N)$ **map wrapped**$(1, 0)$

maps the N rows of A cyclically onto the processors; the specification

double $B[N][N]$ **by block**$(N, N/P)$ **map regular**$(1, 0)$

maps the blocks of N/P columns onto the processors. If there are P processors, the mapping is similar to a HPF block decomposition.

The last specification given in the parameter list concerns the transfer mode for values between the caller and the distributed phase: allowed modes are **IN, OUT** and **INOUT**. This specification is similar to the Fortran90 `INTENT` attribute. Figure 2 shows an example of a distributed phase.

```
#define N 512
#define P 4

dist jacobi(double B[N][N] by block(N,N/P) map regular(0,1) mode INOUT)
 double A[N][N] by block(N,N/P) map regular(0,1);
{
 int i,j;

 for (j=1; j<N-1; j++)
   for (i=1; i<N-1; i++)
     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]);
 for (j=1; j<N-1; j++)
   for (i=1; i<N-1; i++)
     B[i][j] = A[i][j];
}
```

```
SUBROUTINE JACOBI (B)
   INTEGER, PARAMETER :: N = 512
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: B
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A

!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
```

Figure 2: Kernel of the Jacobi algorithm in C-Pandore and HPF

Some other constructs have been added to the C language, with no direct relation with distribution, to improve the ease of programming. Ordinary C functions are not allowed in the Pandore language but in addition to distributed phases, two features are offered to the programmer: *macros* and *closed functions*.

Macro declarations are similar to procedure definitions but parameters are passed "by name" and calls to a *macro* are in-lined by the compiler. *Closed functions* are similar to C functions but cannot access global variables nor modify distributed arrays. Close functions are similar to HPF `PURE` functions.

4 The Pandore Compiler

The compiler produce SPMD code from the user-supplied data decomposition according to the owner-writes rule. Two compilation schemes are embedded in the compiler. For reductions and parallel loops, the compiler applies an optimized scheme [16] performing loop bounds reduction and message vectorization. For statements that cannot be optimized, the compiler relies on the well-known *runtime resolution* technique: masks and communication operations are introduced at the statement level to fetch distant data and select the processor responsible for the computation [5].

Because distributed array management is a critical point to achieve good performances, an original distributed array management based on paging [17] has been developed to support both schemes.

4.1 Compilation of Parallel Loops

For reductions and parallel loops with one statement, where loop bounds and array references are affine functions, the compiler generates a code that comprises two parts: a communication part followed by a computation part. The communication part is in charge of pre-fetching non-local data from other processors. In the computation part, no difference is made between *local computations* (that is those involving only local data) and *non-local computations* (that need to use data received in the communication part). This does not affect the performance of the generated code since our distributed array management provides a uniform and efficient access method for local data and copies of distant data. Besides, performing a separation between local and non-local computations does not seem realistic in the general case, with regard to compilation time and code fragmentation.

Actually, the compiler generates a series of communication codes. One communication code is produced for each right hand side reference to a distributed array and is decomposed in its turn into a *send* part and a *receive* part. Loop bounds and array subscripts but also the distribution of the arrays involved in the computation are analyzed by the compiler: for a given right hand side reference to a distributed array, the associated set of data that must be exchanged between processors is characterized by a *polyhedron* whose scanning [12, 7, 15] constitutes the basis of the SPMD send code and receive code for the reference. The way arrays are represented in the local memories is also taken into account by the compiler so that the data to be moved from one processor to another are scanned in the appropriate direction. This permits the transfer of contiguous zones (both on the sender and the receiver side) and so eliminates the need of coding/decoding and copying between message buffers and local memories. The generation of the SPMD computation code relies on

the same technique: according to the analysis of the left hand side reference, the compiler constructs a polyhedron that defines the set of iterations that must be performed on each processor.

4.2 Management of Distributed Arrays

The distributed array management that completes the two compilation schemes balances the memory requirements and the speed of accesses to local data. It provides a uniform representation for local data and copies of distant data. Each block of a distributed array is decomposed into pages thus an array is represented on a processor by a table of pages that contains both *local pages* (pages of the blocks owned by the processor) and *distant pages* (copies of pages owned by other processors). For a given distributed array, the direction and the size of its pages are determined by the compiler so that the global to local index transformation involves only low level operations (logical shifts and masks) and the size of the table of pages is minimized.

It should be noticed that this technique is different from utilizing a shared virtual memory: the notion of page fault is here irrelevant, a specific paging mechanism is defined for each array and memory allocation and communications are not performed page-wise necessarily.

5 The Jacobi Kernel

The Jacobi Relaxation Iterative Method can be used to approximate the solution of a partial differential equation discretized on a grid. At each time step, the current approximation is updated by computing for each grid point the weighted average of the values of the neighboring points. We focus here on the kernel of this algorithm that consists of two loop nests working on two 2-D arrays A and B. The first loop nest computes in array A the current approximation from the values stored in array B that represents the last approximation. The second one transfers elements of A into B.

The distributed phase corresponding to the kernel is shown in figure 2. Arrays A and B are both distributed into P groups of columns (blocks of size $N \times N/P$), one group on each of the P processors. The mapping is not significant here as there is only one block per processor. With such a distribution, the workload is evenly distributed among the processors and the second loop nest is executed without any communication because arrays A and B are fully aligned. Communication is needed in the first loop nest. Indeed, computing elements of the boundaries of each block necessitates accessing elements situated on the neighboring processors. The symmetry of the accesses would permit a row-wise distribution, leading to the same cost of communication. However, as in both loop nests elements are accessed column-wise (loop j is the outer loop), locality is best exploited with a column-wise distribution[1].

[1] The compilation process does not perturb the loop order when restricting the computation loop domains

As they conform with the conditions under which the polyhedron-based compilation scheme applies, these two loop nests are fully optimized by the PANDORE compiler. In particular:

- Iteration domains are restricted.
- Messages are fully vectorized.
- Direct unbuffered communications are used.
- Index conversions are reduced to the identity function (the page number is the column number and the page offset is the row number).

The performances of the produced code for various input sizes[2] are summarized in table 1. They are almost optimal for small numbers of processors. For a given array size, the number of operations needing only local data performed by a processor is inversely proportional to the number of processors. The boundaries are of fixed size, so performances decrease with large numbers of processors. However, one can notice that performances remain at a good level even with small data sizes: for $N = 128$, the efficiency reaches 67% for 16 processors although 1/4 of the columns are exchanged in the first loop nest.

Although its scope is wider, the joint use of the optimized scheme and the run-time system proves to be as efficient as more classic compilation methods such as the overlap [18].

$N = 128$		
# proc	*time (sec)*	*efficiency*
2	0.242	86%
4	0.128	81%
8	0.068	76%
16	0.039	67%
32	0.027	48%

$N = 256$		
# proc	*time (sec)*	*efficiency*
2	0.970	87%
4	0.495	86%
8	0.253	84%
16	0.132	80%
32	0.075	70%

$N = 512$		
# proc	*time (sec)*	*efficiency*
2	—	—
4	1.961	86%
8	0.999	85%
16	0.503	84%
32	0.261	81%

Table 1: Performances results for the Jacobi Kernel on the Intel iPSC/2

6 Parallelization of a Wave Propagation Application with Pandore

Several tests on classical algorithms such as matrix-matrix multiplication, Gram-Schmidt or LU factorization have already been conducted to evaluate the Pandore compiler [3]; the next step of the validation of the compiler goes through experimentations on real applications.

[2] The memory size on the iPSC/2 nodes did not allow the program execution for $N = 512$ and $P = 2$

6.1 The Wave Propagation Application

We describe here the parallelization/distribution with Pandore of a wave propagation algorithm that has been developed by the French Petroleum Institute (IFP). The application, whose core is about one thousand line long, simulates the wave propagation in a bounded 2D space. Waves are generated by an explosion triggered at a given point of the considered space. The program studies the temporal evolution of the waves at several points of the space where some sensors are located. It takes as input a number of simulation parameters such as the time and space steps, the frequency of the explosion source and the positions of the sensors.

The numerical algorithm follows a discretized finite element method. It corresponds to the second order time discretization ($V(t = n + 1) = F(V(t - n), V(t = n - 1))$) and to the second order discretization of the spatial partial derivatives from the continuous system of PDE describing the waves propagation in an heterogeneous domain.

The results of the algorithm are the values of the horizontal and vertical movements at each time step, for each sensor. The program is divided into two phases:

- The initialization phase: it defines the initial conditions of the explosion and the constraints associated with the nature of the propagation domain.
- The computation phase: this phase consists of a main loop representing the time evolution. At each iteration step we compute the horizontal and vertical movements at time $t + 1$ and $t + 2$, for each point of the grid representing the 2D space.

The final results, i.e. the movements associated with the sensors, correspond to a grid sampling. During the computation phase, for each movement, four arrays are used: U_p and U_m for the horizontal movements and W_p and W_m for the vertical ones.

The body of the loop is composed of four similar parts corresponding to the following computations:

$$\begin{cases} U_p(t+1) & = & f(U_m(t), U_p(t+1)) \\ W_p(t+1) & = & g(W_m(t), W_p(t+1)) \\ U_m(t+2) & = & f(U_p(t+1), U_m(t)) \\ W_m(t+2) & = & g(W_p(t+1), W_m(t)) \end{cases}$$

Functions f and g comprises two series of nested loops. The first one is a series of 2-deep loops operating on the inner part of the grid. These loops are comparable to the first part of the Jacobi kernel presented earlier. The second series is made of several 1-deep loops operating on the upper border of the grid.

6.2 Distribution of the Program

We describe here the steps to transform the initial sequential program into a C-Pandore one. Only one distributed phase is needed for this application, corresponding to the initialization phase followed by the computation phase. So, using the *dist* construct of the

C-PANDORE language to encapsulate this phase, all the computation will be automatically distributed on the nodes of the target parallel computer.

Exchanges between the host (or a dedicated node) and the computing nodes are only performed at the beginning of the distributed phase (to send the simulation parameters) and at the end of the computation (to transfer the final results giving the movements associated with the sensors). These exchanges will be automatically handled by the compiler according to the distribution parameters of the distributed phase.

The body itself of the distributed phase has been slightly modified in order to exhibit parallel nested loops which conform to the conditions under which the compiler performs loop optimization. We obtain that easily for the four main computation parts described in 6.1 because the loops naturally appear as parallel loop nests with affine array references and loop bounds.

The main task when writing the C-PANDORE program resides in the the choice of the array decompositions. The main arrays, described in 6.1, are 2D arrays representing the propagation space grid. They are used together with a tenth of 2D temporary arrays of the same type. Eight other 1D arrays are used for the computation of the border of the grid.

For this algorithm, we chose a column-wise decomposition for all the 2D arrays because

- in the computation of the values associated with the inner part of the grid, the dependencies are similar to those found in the Jacobi, leading to this decomposition as one of the best choices;
- the 1-deep loops operating on the upper border of the grid necessitate a column-wise decomposition in order to distribute the workload;
- the 2-deep loops are column oriented so a column decomposition enforces the locality for the greatest part of the computation.

Given P, the number of processors, each (N,N) array is partitioned into blocks of size $(N,N/P)$. We obtain a C-PANDORE program that, except for the specification of the distribution phase and the partitioning of the arrays, corresponds almost exactly to the sequential one.

6.3 Performance Results

6.3.1 First Results

We ran the above described version of the Wave Propagation program on the IPSC/2. The performance results are given in table 2. The overall performances are satisfactory, considering that the parallel code has been produced automatically. With a good adequation between the data size and the number of processors, an efficiency around 70% is reached. Furthermore, it can be noticed that, even for small arrays, efficiency decreases slowly when adding processors.

$N = 128$		
# proc	*time (sec)*	*efficiency*
2	136	75%
4	71	72%
8	38	66%
16	22	56%
32	17	38%

$N = 256$		
# proc	*time (sec)*	*efficiency*
2	—	—
4	280	71%
8	145	69%
16	77	65%
32	42	59%

$N = 512$		
# proc	*time (sec)*	*efficiency*
2	—	—
4	—	—
8	—	—
16	289	68%
32	150	66%

Table 2: Performance results for the Wave Propagation Application

6.3.2 Further Optimizations

Sampling Associated with the Sensors

In the C-PANDORE program obtained in section 6.2 the parallel loops performing the sampling lead to communications that could be avoided. This is due only to the fact that alignment cannot be expressed yet in C-PANDORE; hence, the arrays storing the movements associated with the sensors are not aligned with the arrays representing the space grid. However, a trivial manual renumbering of the sensors points is possible to make the sampling arrays aligned with the grid arrays. After this transformation, the sampling loops are executed without communication. The performances are not very much affected by this modification as the sampling only represent 5% of the total execution time.

Avoiding Multiple Transfers

During execution some array elements are sent several times to the same processor without having been modified. These unnecessary transfers may be avoided by storing the data in auxiliary arrays after the first send. We experimented this –non trivial– optimization which necessitates to declare new arrays. The performances are improved by about 5% of the total execution time. This does not appear as significant a gain, considering that for this application the memory cost is of great importance. In fact, the size of the memory on each node of the parallel computer severely limits the experiments that may be conducted.

7 Conclusion

We have shown in this paper that the PANDORE compiler allows distributing efficiently a real application, without any significant effort from the programmer. Indeed, the first Pandore source of the Wave Propagation program presented in the paper is very similar to the sequential original program and produces correct performances. This tends to confirm the viability of the data-parallel approach for scientific computing provided that general enough and non-naive compiling and run-time techniques are applied.

To handle very large applications, in particular applications that comprise multiple modules or necessitate intensive I/O, other techniques must be integrated to existing environ-

ments. For this purpose, the joint study of redistribution, procedures and separate compilation is under way in the PANDORE project.

References

[1] S. M. Amarasinghe and M. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[2] F. André, O. Chéron, and J.-L. Pazat. Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 293–308, Elsevier Science Publishers B.V., 1993.

[3] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. *The Pandore Compiler: Overview and Experimental Results.* Research Report 869, IRISA, October 1994.

[4] P. Brezany, B.M. Chapman, and H.P. Zima. *Automatic Parallelization for GENESIS.* Research Report ACPC/TR 92-16, Austrian Center for Parallel Computation, November 1992.

[5] D. Calahan and K. Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, October 1988.

[6] 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.

[7] 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.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[9] J. Grosch and H. Emmelmann. *A Tool Box for Compiler Construction.* Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, January 1990.

[10] F. Guidec and Y. Mahéo. *POM: a Parallel Observable Machine.* Technical Report, IRISA, 1994. To appear.

[11] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8), August 1992.

[12] 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.

[13] 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.

[14] L. Jerid, F. André, O. Chéron, J.-L. Pazat, and T. Ernst. *HPF to C-Pandore Translator.* Technical Report 2283, INRIA, Mai 1994.

[15] M. Le Fur. *Parcours de polyèdre paramétré avec l'élimination de Fourier-Motzkin.* Technical Report 858, IRISA, Rennes, France, September 1994.

[16] M. Le Fur, J.-L. Pazat, and F. André. *Static Domain Analysis for Compiling Commutative Loop Nests.* Research Report 2067, INRIA, September 1993.

[17] Y. Mahéo and J.-L. Pazat. *Distributed Array Management for HPF Compilers.* Research Report 2156, INRIA, December 1993.

[18] H. P. Zima, H.-J. Bast, and M. Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, (6):1–18, 1988.