PANDORE : a system to manage data distribution

Françoise André, Jean-Louis Pazat, Henry Thomas

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports de Recherche

N° 1195

Programme 2
Structures Nouvelles d'Ordinateurs

PANDORE : A SYSTEM TO MANAGE DATA DISTRIBUTION.

Françoise ANDRÉ
Jean-Louis PAZAT
Henry THOMAS

Mars 1990

RR_1195

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n° 519
Février 1990 - 14 Pages

Pandore:
A System to Manage Data Distribution.

Françoise André *, Jean-Louis Pazat †, Henry Thomas ‡

20 février 1990

Abstract

The goal of the Pandore system is to allow the execution of parallel algorithms on DMPC (Distributed Memory Parallel Computers) without having to take into account the low-level characteristics of the target distributed computer to program the algorithm. No explicit process definition and interprocess communications are needed. Parallelization is achieved through logical data organization. The Pandore system provides the user with a mean to specify data partitionning and data distribution over a domain of virtual processors for each parallel step of his algorithm.

At compile time, Pandore splits the original program into parallel processes which will execute the same code over the different parts of the data, according to the given decomposition. In order to achieve a correct access to the data structures distributed over the processors, the Pandore system provides a communication layer, which is an abstraction of a message-passing architecture. This intermediate level is then implemented using the effective primitives of the real architecture (in our specific case ,an Intel iPSC/2).

Main topic: logical data partitionning and distribution;compilation for Distributed Memory Parallel Computers .

Pandore :
Un Système de Gestion de Données Distribuées

Résumé

L'objectif du système Pandore est de permettre l'exécution d'algorithmes parallèles sur des machines à Mémoires Distribuées, sans contraindre l'utilisateur à programmer son algorithme en tenant compte des caractéristiques techniques du calculateur cible. En particulier les processus et les communications inter-processus sont définies de manière implicite.

La parallélisation du programme est réalisée en fonction d'une organisation logique des données qui se prête à distribution. A l'aide d'un système d'annotations, l'utilisateur spécifie le partitionnement et la distribution de ses structures de données sur la Machine Virtuelle adaptée à chaque phase de son algorithme.

Pandore compile le programme source sous forme de processus parallèles exécutant le même code sur les différentes parties de données distribuées. Les communications nécessaires pour accéder aux données distantes sont ajoutées au code sous forme d'appels de primitives indépendantes de toute machine précise. Avant exécution, ces primitives sont traduites en terme des opérations de communication réelles.

*e-mail: fandre@irisa.fr
†e-mail: pazat@irisa.fr
‡e-mail: hthomas@irisa.fr

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Today Distributed Memory Parallel Computers (DMPCs) such as networks of Transputers, NCUBE computers or INTEL iPSC, are commercially available. Unfortunately, associated parallel programming methodologies are missing and the process of writing a distributed parallel program is, most of the time, a very hard task.

Traditionally, the user proceeds through different steps:

- Firstly, he finds a method for splitting his data into pieces in order to work in parallel on them.
- Secondly, he maps the previous logical decomposition onto the DMPC. In this phase, the user tries to optimize the use of resources in order to obtain the best performances from his computer. Clever load balancing and process scheduling are often needed.
- Endly, he has to implement data accesses which are of two kinds: local or distant. In that last case message communication is used as this is the only mean currently available on most DMPCs. The division of memory among processors in DMPCs represents the major difficulty the programmer has to tackle with. The problem is to know the physical location of the data and to synchronize the processes in order to maintain up-to-date and consistent values for shared data.

As all these phases are not realized automatically, the user may need to prove some properties of his program —like the deadlock–free property— both at the algorithmic level and at the implementation level. All this programming work goes far beyond the expression of an algorithm in some programming language: it requires a deep comprehension of parallelism and a fairly good knowledge of the DMPC used. Moreover, the final program is fully machine dependent.

We believe this is presently a severe limit for the use of DMPCs.

Among these phases, the user is primarily concern by the logical decomposition of his application whereas the other steps are mainly technical ones. Presently, logical decomposition through automatic partitionning seems to be intractable in most cases, whereas we think that automatic tools could be defined to solve most of the problems related to physical mapping and data accesses, freeing the user of the low–level implementation details.

After having given a global view of the Pandore system in section 2, we detail the Pandore annotation language in section 3. Section 4 describes the actual intermediate code generation. We conclude with a comparison with other related works and with the presentation of the main developments that are envisaged to improve the current version.

2 Overview of the Pandore system.

The main field of applications for DMPCs is scientific programming which involves large data structures such as matrices. To parallelize these applications on DMPCs it seems more efficient to rely on methods based on data partitionning rather than on code analysis such as vectorization techniques, which exhibit a too fine grain parallelism.

According to these remarks our approach is based on data distribution which will enhance spacial locality of reference. The Pandore system provides the user with a mean to logically specify data

partitionning. Then it automatically generates distributed processes and manages data distribution through consistent local or distant accesses.

In the Pandore system the user task is to annotate his data structures in order to define a regular decomposition on them. For example, assuming `Win` is a vector of N elements, the statement

```
partition(Win,Block(K))
```

partitions the array Win in $\lceil N/K \rceil$ subarrays of length K.

The user specifies also a virtual processor domain which is called the Virtual Distributed Machine (VDM). For example the declaration:

```
processor row[4]
```

defines a virtual machine of 4 processors connected as a linear array.

A VDM corresponds to the ideal parallel architecture for a given computional block. Such a block of instructions is prefixed with the VDM's name. The partitionned data which are used during the block execution and so should be distributed over the current Virtual Distributed Machine by the Pandore system are listed at the beginning of the block. For example,

```
vdm row (Win,IN).
```

opens the VDM `row` and distributes `Win` upon `row`.

Figure 1 provides an overview of the Pandore system. In the first step the Pandore Compiler produces a parallel program which consists of a set of processes. Each process executes the same code on different parts of the data according to the given decomposition. To achieve correct access to the variables distributed over the processors, the Pandore system provides a communication layer, which is an abstraction of a message–passing mechanism. The resulting object code is written in an intermediate language (Virual Distributed Machine Language VDM_L) which may then be translated into a specific machine code. The following sections explain the different steps in more depht.

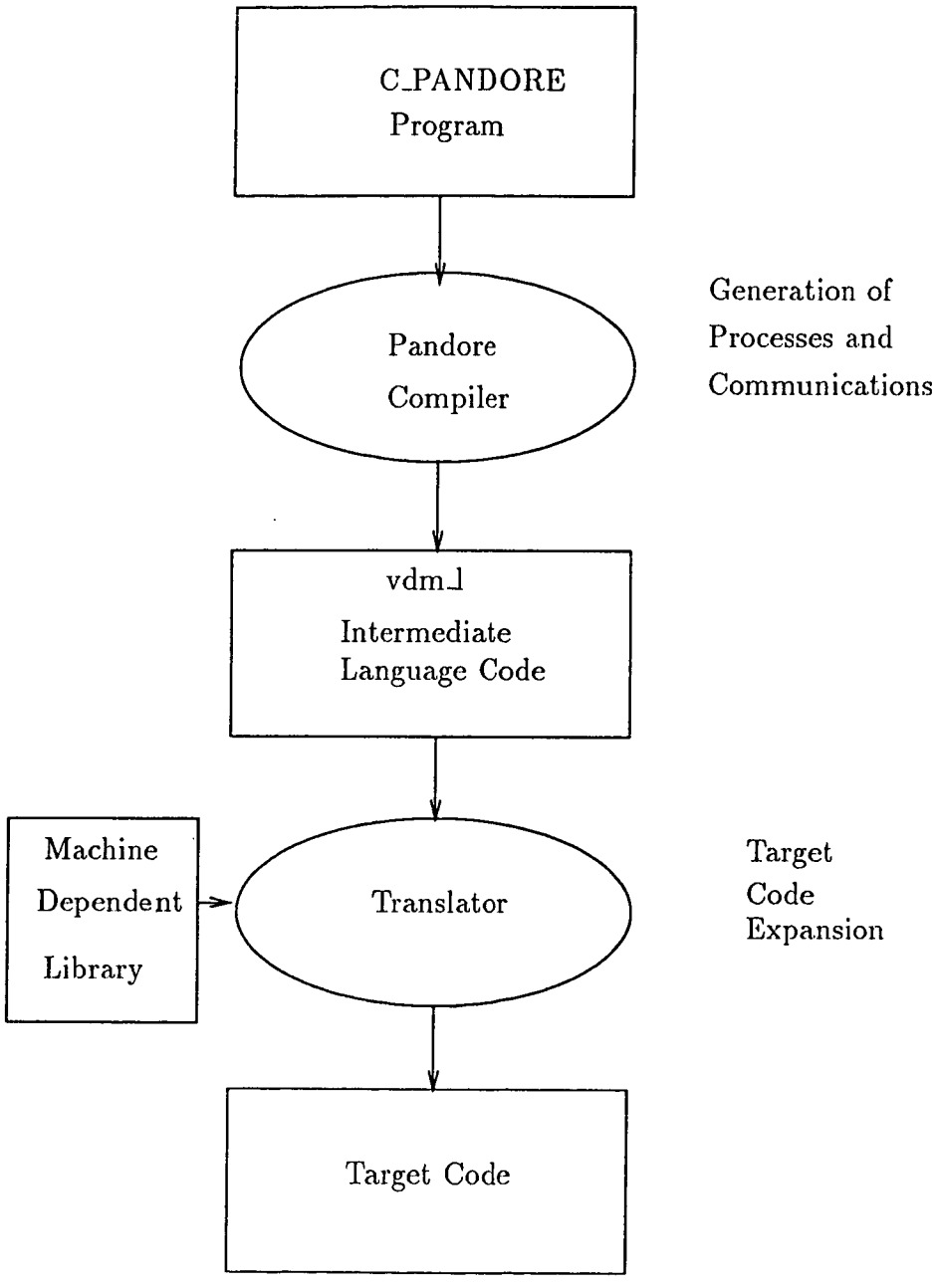

Figure 1: The Pandore System

3 The Pandore Language

A source language for the Pandore system can be built upon any sequential language (Pascal, Fortran, C,...). For our first investigation, we have implemented Pandore upon a subset of the C language (C_Pandore) with new keywords and constructs in order to express:

- The partitionning of data structures,
- The definition of Virtual Distributed Machines (VDM),
- The distribution of a data structures over VDMs,
- The non sequential loops (`forall` loops).

3.1 Data partitionning

If we want to partition and distribute a variable, it must be declared as "distributed", for example:

```
distributed int T[N][M]
```

declares an array of `N x M` integers which may later be partitionned using the statement:

```
partition(T,Block(k,l))
```

The `partition` statement allows to decompose an n-dimensional array into n-dimensional sub-blocks using the `Block` function. For example the 2-dimension array T is partitionned into blocks of size $k \times l$. If N mod $k \neq 0$, the last blocks in the first dimension are of size N mod k. If M mod $l \neq 0$, the last blocks in the second dimension are of size M mod l.

Special cases are

- $k = 1$: The array is partitionned into rows.
- $l = 1$: The array is partitionned into columns.

In numerous applications, the computation associated with one sub-block often refers to boundary elements of the other sub-blocks. So we provide a `block_overlap` function, which allows to duplicate boundary slices of neighboring sub-blocks. The elements of the duplicate slices are read-only. For each dimension, the overlap parameters specify:

- The size of the slice of the previous block in the current dimension which is duplicated,
- The size of the slice of the next block in the current dimension which is duplicated,
- The optional overlap for the extreme blocks.

For example, the statement

```
partition(T,Block_overlap((k,1,1,None)(1,1,1,None)))
```

partitions the array `T` into blocks of size `k x 1` with an overlap of a slice of size one, in each direction except for the extreme blocks.

Presently we only offer these two partitionning functions. Extensions are envisaged for the next version of the system.

3.2 Virtual Distributed Machines

In C_Pandore, a VDM is a set of interconnected virtual processors which must be declared like other variables before being used.

The keyword `processor` introduces a new type which allows to define VDMs as arrays of processors. A qualifier is used to specify particular topologies. At present it is only possible to define the following VDM topologies: vector, ring, grid and torus. For instance:

- `processor my_vector[N]` declares a linear array of `N` processors
- `torus processor my_ring[N]` declares a ring of `N` processors
- `processor my_grid[N][M]` declares a grid of `NxM` processors
- `torus processor my_torus[N][M]` declares a torus of `NxM` processors.

A VDM is in use for the duration of one computational phase corresponding to the execution of a parallel block. Only one VDM may be "opened" at a time; if no VDM is explicitely opened, the current VDM is a single processor. The statement

```
vdm my_vdm(<parameters>){<parallel block>}
```

makes available (opens) the virtual machine `my_vdm` as a target machine for the distribution of the data structures specified in `<parameters>` and for the execution of the block of instructions.

3.3 Data distribution over VDMs

The data distributed over the current VDM are indicated at the opening of the VDM as a list of items ((`<variable>`,`<mode>`)) .

The first parameter of an item specifies the name of the variable to distribute and the second parameter (`IN`, `OUT` or `INOUT`) indicates:

- that the values of the distributed structure are significant at the entry of the block (`IN`) and that the structure will recover the same values at the end of the block ,
- that the modified values of the structure are significant after the end of the block (`OUT`),
- or that they are significant before and after the block execution (`INOUT`).

The distribution is realized by the Pandore system in the following way :

- If the variable is a partitionned array:
  The distribution of the data structure is chosen according to the topology of the current VDM. If there are less blocks in the partitionned array than processors in the VDM, only the first processors of the VDM will receive a block. If the partitionned array is too large, block assignation is made by folding (on vector and grid VDMs) or in a modulo fashion (on ring and torus VDMs).

  The partitionning mode for the variable should be given before the opening of the block. No partitionning statement is allowed inside a VDM.

- If the variable is of a basic data type (int, float, ...):
  only the following distribution modes are available:
  - `(x:IN)`: Each processor owns an independent copy of the variable. The variable will retrieve its initial value at the end of the block.
  - If a variable is declared inside a VDM, each processor owns an independent copy of the variable .This is the case for loops control variables such as `i` in the statement `forall(i;1;N-2)` .
  - `(x:OUT)` or `(x:INOUT)`: In that case a single processor (chosen by the system) owns the variable.

In all cases the value of the `<mode>` parameter allows to know if it is necessary to distribute the actual content of the data structure or not.

The data distribution induces the location of the execution of each instruction: an instruction which modifies an element of a structure is executed on the processor which owns that element. So, write accesses are local whereas read accesses may be local or distant. Within an instruction, the dependance between data elements is generally expressed by their neighborhood in the VDM, but this is not restrictive.

For example, if the arrays `Win` and `Wout` are distributed such that each element of the arrays is on a processor of a linear VDM, the statement:

```
Wout[i] = a * Win[i-1] + b * Win[i] + c * Win[i+1];
```

is executed on the processor `k` which owns `Wout[i]` (it also owns `Win[i]`), the values of `Win[i-1]` and `Win[i+1]` are collected from the neighbor processors `k-1` and `k+1`. Using the same distribution, it is also possible to execute the statement:

```
Wout[i] = a * Win[i-2] + b * Win[i-1] + c * Win[i] + d * Win[i+1] +e * Win[i+2];
```

This statement is also executed on the processor `k` which owns `Wout[i]`, the values of `Win[i-2]` and `Win[i+2]` are collected from the processors `k-2` and `k+2` even though these processors are not neighbors of processor `k` in the VDM.

Figure 2 gives a short example of a C_Pandore program corresponding to the computation of the convolution product defined as : $Wout_i = a\,Win_{i-1} + b\,Win_i + c\,Win_{i+1}$

```
/* C_Pandore Convolution Program */

#define N 1000
distributed int Win [N];
distributed int Wout[N];

distributed int a,b,c;
int k;

processor row[4]; /* VMD architecture: row of four processors */

main()
{
  /* get the initial values from user        */
  printf("Enter coefs");
  scanf("%d%d%d",a,b,c); }
  for (k=0; k<N; k++) {
    printf("Enter the element #%d of Win ",k);
    scanf("%d",Win[k]); }

  /* parallel phase */

  /* Win is partitionned in blocks of length N/4 */
  /*   overlap is of 1 element in each direction       */

  partition(Win , Block_overlap(N/4,1,1,None));

  /* Wout is partitionned in blocks of length N/4 with no overlap */
  partition(Wout, Block(N/4));

  VDM row((Win ,IN),(Wout,OUT),(a,IN),(b,IN),(c,IN))
  {
    int i;
    forall(i; 1; N-2)
      Wout[i]= a*Win[i-1] + b*Win[i] + c*Win[i+1];
  }

  /* now print the results  */
  for (k=1; k<=N-2; k++)
    printf("Wout [ %d] = %d \n", k,Wout[k]);
}
```

Figure 2: Convolution Program in Pandore

4 Processes and communications generation.

Provided with the source program, the Pandore system generates a vdm_l program which is a set of communicating processes. Each virtual processor of a VDM runs a slave process which is uniquely identified in the system. Slave processes are mastered by a *master* process which runs "outside" the VDM.

The original user code is replicated in every slave process; communications and synchronizations are added in order to insure consistent access to distributed shared data. Each statement is guarded so that all write accesses to variables are local to the processor which owns the variable. Non local read accesses are achieved with a non blocking message passing mechanism. The purpose of the master process is to execute the sequential part of the vdm_l program and to distribute and collect the data.

The vdm_l program is machine independent. Presently vdm_l is implemented as a set of C programs making calls to the Pandore communication library. Before execution, the vdm_l program should be translated into the real machine code. We are writing an implementation of the Pandore library for the iPSC2.

This is a very simple run-time resolution scheme which certainly will give poor performances. As indicated in the conclusion, we now plan to work on compile-time optimizations.

5 related work

Most of the systems currently in use for DMPCs require explicit parallel processes definition. Some improvement concerning the communications is provided through the use of a global memory abstraction as for instance in [1] and [2].

We here focus on systems which allow implicit process definition.

Superb [3] is the parallelizer for the Suprenum project. It provides the user with a set of interactive tools to partition the data structures and to transform the program into parallel subtasks. The data partitionning takes into account the size of the application, the size of the machine, the range of loops and the dependencies. As this approach is based on a user-oriented dialog and on a catalog of parallelization transformations it differs quite significantly from ours.

The following projects are based on specific language constructs to express distribution as in Pandore.

Dino [4] is a language where it is possible to define "environment structures" which are arrays of processors (very similar to our VDMs). Data are explicitely distributed on an environment structure. Concurrency is achieved by the use of "composite procedure calls" standing for a concurrent call of one instance of a procedure on each processor. Arguments are passed by value (IN) or by result (OUT) or both. In Dino the programmer must explicitely express that an access is distant and he must choose the synchronization modes for replicated variables. This is the main difference with our system.

Id_nouveau [5] is a functional language with array constructs added. It allows explicit distribution and replication of data structures on the real machine. The run-time resolution used is very similar to ours.

In Parascope [6], data partitionning and distribution is achieved by the use of explicit index functions. The partitionning of an array defines a virtual array which can later be distributed on the real architecture.

In the KALI project [7], the data partitionning and distribution is defined at the declaration of the variable and may not be changed. Special keywords are used to express the partitionning modes, and the syntax is based on a Pascal-like language.VDMs are defined as arrays of processors, the size of which is computed during the loading phase of the program on the real architecture. The parallel execution scheme associates each iteration of a loop with one processor. On the contrary, we distribute each execution of a statement on the processor which owns the modified variable.

All these projects are working along the same idea, using different methods or putting the emphasis on one aspect of the problem or the other. Compared with most of them, we have a block-oriented approach for distribution and VDM utilization; we also try to be as far as possible machine independent in order to be portable on different DMPCs.

6 Conclusions

We believe that our approach is attractive for the user of distributed memory computers as the programming task is much easier, allowing him to concentrate on the logical properties of his algorithm.

But this facility will be appreciated only if the performances that are obtain by automatic code generation are comparable to the ones obtain by manual coding. We do not forget that performance improvement is the main goal of the user of DMPCs.

After having realized a first version of the Pandore system which uses a very crude compiling technique, we now have to improve our compiler. A lot of optimizations are planned ranging from static generation of communications with anticipation, to fully automatic partitionning for typical data structures. This is now our main research axis, the two others being to broaden the applications field (that necessitates to manage different data structures such as trees) and to improve the execution scheme on real architectures.

References

[1] Geoffrey C. Fox and Woitek Furmansky. Communication algorithms for regular convolutions and matrix problems on the hypercube. In *Hypercube Multiprocessors*, pages 223–238, 1987.

[2] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in linda. In *Thirteenth Annual ACM Symposium on Principles of Programming Langages*, 1986.

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

[4] Matthew Rossing, robert B. Schnabel, and Robert Weaver. Dino: summary and examples. In *3' Conference on Hypercubes Concurrent Computers and Applications*, pages 472–481, 1988.

[5] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, ACM, June 21–23 1989.

[6] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, (2):151–169, 1988.

[7] Charles Koebel and Piyush Mehrotra. *Supporting Shared Data Structures on Distributed Memory Architectures.* Technical Report csd-tr 915, Department of Computer Science, Purdue University, 1990.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 512 **A FAULT TOLERANT TIGHTLY COUPLED MULTIPROCESSOR ARCHITECTURE BASED ON STABLE TRANSACTIONAL MEMORY**
Michel BANATRE, Philippe JOUBERT
Février 1990, 20 Pages.

PI 513 **BUILDING A GLOBAL TIME ON PARALLEL MACHINES**
Jean-Marc JEZEQUEL
Février 1990, 28 Pages.

PI 514 **PARALLELISATION D'UN RESEAU NEURONAL**
Krysztof WOLINSKI
Février 1990, 20 Pages.

PI 516 **COMMENT INTRODUIRE LA CONTIGUITE EN ANALYSE DES CORRESPONDANCES ? Application en segmentation d'image.**
Brigitte ESCOFIER, Habib BENALI, Kaddour BACHAR
Février 1990, 26 Pages.

PI 517 **MACHINE MODELING AND LOOP OPTIMIZATION FOR HORIZONTAL MICROCODED MACHINES**
François BODIN, François CHAROT
Février 1990, 24 Pages.

PI 518 **MULTISCALE SYSTEM THEORY**
Albert BENVENISTE, Ramine Nikoukhah, Alan S. Willsky.
Février 1990, 30 Pages.

PI 519 **PANDORE : A SYSTEM TO MANAGE DATA DISTRIBUTION**
Françoise ANDRE, Jean-Louis PAZAT, Henry THOMAS
Février 1990, 14 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique