Efficient code generation for distributed memory machines

Françoise André, Olivier Chéron, Jean-Louis Pazat, Henry Thomas

UNITÉ 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° 1522

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

EFFICIENT CODE GENERATION FOR DISTRIBUTED MEMORY MACHINES

Françoise ANDRÉ
Olivier CHÉRON
Jean-Louis PAZAT
Henry THOMAS

Septembre 1991

RR-1522

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99.84.71.00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Efficient Code Generation for Distributed Memory Machines *

Françoise André, Olivier Chéron, Jean-Louis Pazat, Henry Thomas †

July 1991

Programme 1

Publication Interne n° 596 - 14 pages

Abstract

Although several new languages based on parallel processes are well suited to program parallel architectures, most of the scientific programmers still prefer to use the well-understood sequential model. Therefore, parallelization of sequential programs for Distributed Memory Machines (DMMs) is an important research axis. We present the PANDORE system whose goal is to allow a parallel execution of programs on DMMs without requiring any deep knowledge of such machines from the user.

Specific constructs used by the programmer inside a sequential program, indicate how to partition the data manipulated by this program. These constructs help the compiler to split the original program into processes in the parallelization phase. According to the SPMD model (Single Program, Multiple Data), these processes perform the same code over the different parts of the data. The generated processes are expressed in a machine independent intermediate language. Machine specific libraries are used to produce the code for the target computer.

Presently, a prototype compiler has been realised. The source language is based upon a subset of the C language extended with partitioning features. In the first implementation of the compiler, the efficiency of the produced code was not satisfying, due to the use of a pure SPMD model. Several improvements have been defined. A restricted set of optimizations bound to intermediate code restructuring, has been implemented with satisfactory results in our compiler.

*to appear in the "Parallel Computing 91" proceedings

†e-mail: fandre@irisa.fr, ocheron@irisa.fr, pazat@irisa.fr, hthomas@irisa.fr

Génération de code efficace pour Machines à Mémoires Distribuées

Françoise André, Olivier Chéron, Jean-Louis Pazat, Henry Thomas

Juillet 1991

Programme 1

Résumé

Bien que les langages basés sur la notion de processus communicants correspondent mieux au modèle de programmation des architectures parallèles, la plupart des programmeurs scientifiques préfèrent encore utiliser des langages séquentiels qui leur sont plus familiers. La parallélisation de programmes séquentiels pour les machines à mémoires distribuées (MMD) est donc un axe de recherche important. Nous présentons ici le système PANDORE dont l'objectif est de permettre l'exécution parallèle de programmes sur MMD, sans pour autant requérir de la part de l'utilisateur une connaissance approfondie de ce type de machine.

L'utilisateur annote son programme séquentiel avec des instructions spécialisées précisant comment décomposer et répartir les données. Ces instructions aident le compilateur à générer des processus communicants exécutant le même code sur des données différentes selon le modèle SPMD (Single Program — Multiple Data). Ces processus sont décrits dans un langage intermédiaire, indépendant d'une architecture donnée.

Un prototype du compilateur a été conçu et réalisé ; le langage source est un sous-ensemble de C augmenté de primitives de partitionnement et de distribution. Dans la première version du compilateur, l'utilisation du modèle SPMD strict engendre un code dont l'efficacité n'est pas satisfaisante. Plusieurs améliorations ont été définies et un sous-ensemble de celles-ci a été mis en œuvre avec des résultats encourageants.

1 Introduction

Programming DMMs is a hard work as the user has to cope with parallelism and distribution. In order to facilitate the use of such machines for the application programmers, it is necessary to hide the implementation details as much as possible. One way is to spare the programmers the writing of communicating processes with explicit message passing. To do so, a sequential language is used, leaving much work to the compiler, such as the physical distribution of data structures and code, and the management of distributed data accesses. In many approaches based on this principle, the compiler is guided by a user-defined data decomposition.

In this paper we describe a compiler which uses this method in order to generate parallel processes from a sequential program. After the presentation of closely related projects, the paper discusses the main characteristics of the PANDORE system and focuses on the optimization problems. First, we present a review of our system: the source language, the compilation scheme and the execution scheme. Second, we illustrate its key features by giving a simple example. Third, we present a study of some preliminary results that brings out the primordial improvements that have to be done. Finally, we end with a short description of our ongoing work.

2 Related Work

Traditional parallelization tools for shared memory parallel computers achieve code parallelization through the analysis of the control flow of a program. These compilers try to detect parallelism in the source code but do not take into account any data distribution.

Programming models other than the message–passing MIMD model have been recently investigated for programming DMMs. In all cases the major issues are to provide a simple programming model and to allow the user to express local and distant accesses in the same way.

The models being investigated refer to well known programming methodologies as implicit parallelism (equational model, logic or functional models), sequential model, SPMD model, or SIMD model.

In image processing, for example, the SIMD model is often a natural way to express algorithms. Hatcher et al. [1] have implemented C* (the language used on the connection machine) on an iPSC/2 and Reeves et al. [2] are working on the implementation of an SIMD object oriented language including user specified data partitioning.

The SPMD model without explicit communication between processes is being studied by many research groups. The DINO language [3] allows accesses to distant data through local copies without explicit message passing nor hand written synchronizations insuring the coherence of the different copies. The KALI language [4] offers a special parallel construct (**forall**) for defining distributed iterations among a domain. In KALI, each iteration of a loop is executed on one processor specified by the user according to the data distribution.

As the sequential model is well known to many programmers, much research is developed in this field. New imperative language like BOOSTER [5] have been defined in order to fit data accesses to distributed executions. In this language, data and index domains are separately represented by "shapes" and "views" avoiding the use of indirect accesses to array elements. Moreover the view concept allows to define the data distribution and the compilation scheme.

Other works such as SUPERB [6], FORTRAND [7] and our project PANDORE [8] are based on an existing sequential language (mainly Fortran or C) augmented with data distribution capabilities.

3 An overview of the PANDORE system

The aim of the PANDORE system is to allow the user to run sequential programs on a Distributed Memory Machine. The imperative language, C-PANDORE, includes three special features that the programmer can use to underline the decomposition of the problem he wants to solve.

The first feature is the specification of data decomposition. This specification helps the compiler to distribute the data manipulated by the program over the different processors of the target machine.

The second feature is the specification of virtual processor domains. These domains, called "Virtual Distributed Machines" (VDM for short), represent what the user thinks to be the ideal parallel architectures to run the different phases of his algorithm. A domain defines the inter-processor neighbourhood relations.

The third feature is a special block constructor that is used to link data to a particular VDM. When opening a **vdm** block, the user must specify two things :

- what are the data manipulated by the statements contained in this block,
- what is the VDM that is the most suitable to perform the calculation phase enclosed in this block.

All other instructions of C-PANDORE are those that can be found in any usual imperative language: assignments, tests, loops,...Notably, data accesses are achieved through a global name space.

Our compiler generates a SPMD program by almost replicating the initial code and by including communication statements where remote accesses are performed. Each processor executes conditionally the original code. These conditions are calculated in the sight of the data distribution among virtual processors. Locality of writes is the rule that governs this calculation. This means that an assignment is run by the processor on which the left hand side variable is located. To deal with remote read accesses, our compiler translates each statement S, contained in a **vdm** block, into two intermediate code instructions that are called **Refresh** and **Exec**. Let V be the set of variables that S refers to and P be a set of processes:

- **Refresh**(V, P) updates the values of all the variables in V on all the processes in P.
- **Exec**(S, P) all the processes in P execute the statement S

After the **Refresh** phase, all the necessary values for the execution of S are available locally on each process that must execute the statement S. During this phase, inter-process communication could have been generated.

The **Exec** phase leads S to be executed solely on the processes that own the variables that are modified (assigned) in the statement S (statement masking).

The straightforward (but naive) run-time implementation used in the prototype of our compiler is shown bellow:

```
Refresh(V, P)  ≡   if myself ∈ own(V)
                      then send(P\own(V), V)
                   if myself ∈ P\own(V)
                      then recv(own(V),V)

Exec(S, P)     ≡   if myself ∈ P
                      then S
```

Where

- **myself** is the name of the current process,
- own(V) is the set of processes where data in V are mapped,
- send(Q,W) sends the values of the variables in W to the processes in Q ("non-blocking send": processes in own(W) do not wait for values in W to be received on Q),
- recv(Q,W) waits for receiving values of variables in W from processes in Q.
- communication between processes is FIFO, no messages are lost,

Of course, efficient program distribution necessitates that the user carefully designs his data structures and their partitions and that the compiler optimizes the code in order to avoid unnecessary operations. Some optimization rules and improvements for the PANDORE compiler are described in section 4 and in [9]. Now we are going to explain our compilation and execution schemes in regard to a simple example.

```
distributed int A[100], B[100] ;
processor P[10] ;
. . .
partition A with block(10) ;
partition B with block(10) ;

vdm P ( (A,INOUT) , (B,IN) ) {
   A[0] = B[5] ;                    (S1)
   A[9] = B[95] ;                   (S2)
}
printf("A[0] = %d\nA[9] = %d\n\n",A[0],A[9]) ;
. . .
```

Figure 1: A simple example: the source code

In figure 1 we declare two distributed arrays, A and B, of 100 integers and a virtual processor domain, P, which is a row of 10 processors.

The two **partition** statements specify a decomposition of A and B into parts of 10 elements.

The **vdm** statement associates the A and B data domains which just have been decomposed, with the virtual processors of P. The **INOUT** mode indicates that A will be modified by the statements of the **vdm** block, and the **IN** mode indicates that B will stay the same after the execution of this **vdm** block. The third usable mode is **OUT**. Its semantics ensures that the value of the associated data domain will be significative only at the closing of the **vdm** block[1].

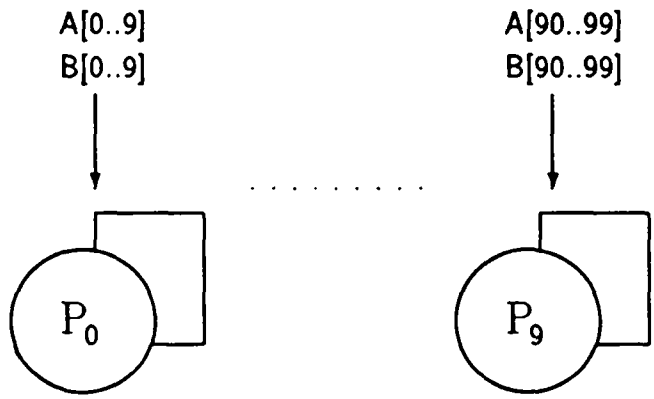

Figure 2: A simple example: the mapping of data

Figure 2 illustrates the mapping of data once the **vdm** block is opened. Let P_i be the virtual processor P[i]. Each P_i receives 10 elements of A and 10 elements of B and stores them in its local memory. Though, A[0..9] and B[0..9] are local to P_0 and A[90..99] and B[90..99] are local to P_9.

The 2 statements S1 and S2 that compose the body of the **vdm** block are translated as shown below:

```
S1    ---->    Refresh(B[5], own(A[0])) ;      (R1)
               Exec(A[0]=B[5], own(A[0]))      (E1)

S2    ---->    Refresh(B[95], own(A[9])) ;     (R2)
               Exec(A[9]=B[95], own(A[9]))     (E2)
```

In the sight of the data mapping we assume that own(A[0]) is P_0 and that own(B[5]) is also P_0. Then, the execution of the first **Refresh** statement, R1, will generate no communication, and the first instruction mask, E1, implies that P_0 will be the only process to execute the assignment: A[0]=B[5].

The second **Refresh** statement, R2, has to deal with two different virtual processors: the owner of A[9] which is P_0 and the owner of B[95] which is P_9. The execution of R2 results in P_9 sending the

[1] All the informations enclosed in the mode-parameters are used by the compiler to avoid unnecessary data transfers between the host and the nodes of the target machine on which the code is to be executed.

value of B[95] to P_0. At the receipt of this value, P_0 will be able to perform locally the assignment: A[9]=B[95] and E2 ensures that P_0 will be the only process to execute this assignment.

Now having depicted our compilation and execution schemes, we present an analysis of our first results and expose some important optimizations that must be implemented to enhance our system.

4 Preliminary results and Planned Improvements

Let us first illustrate what slows our code, considering a simple code which generates no communication.

```
distributed int A[N], B[N], C[N];
partition (A, block(T));
partition (B, block(T));
partition (C, block(T));
...
    for (i=0; i<N; i++)
        C[i]=A[i]*B[i];
```

The code generated by the compiler is the following:

```
distributed int A[N], B[N], C[N];
for (i=0; i<N; i++)
    Refresh(A[i], own(C[i]));
    Refresh(B[i], own(C[i]));
    Exec(C[i]=A[i]*B[i], own(C[i]));
```

The execution time for the code generated by the PANDORE compiler is compared to the execution time of a hand-parallelized code on figure 3. We notice that the PANDORE generated code is approximately limited by a 400ms time bound.

The results of this execution can be explained in two points

- **Refresh** computation: the **own()** function is evaluated at runtime. This induces an increase in execution time. We shall also point that data accesses (C[i]=...) by themselves carry a time overhead: the global index i has to be transformed into a local index on each processor (this transformation is also performed at execution time). This explains the *value* of the bound.

- **Exec** computation: on the example, each processor executes the integrality of the SPMD code, thus each processor executes N iterations, although that, given the data decomposition, only N/T iterations lead to effective computations. This explains *why* we have a bound on the execution time, whatever how many processors we use.

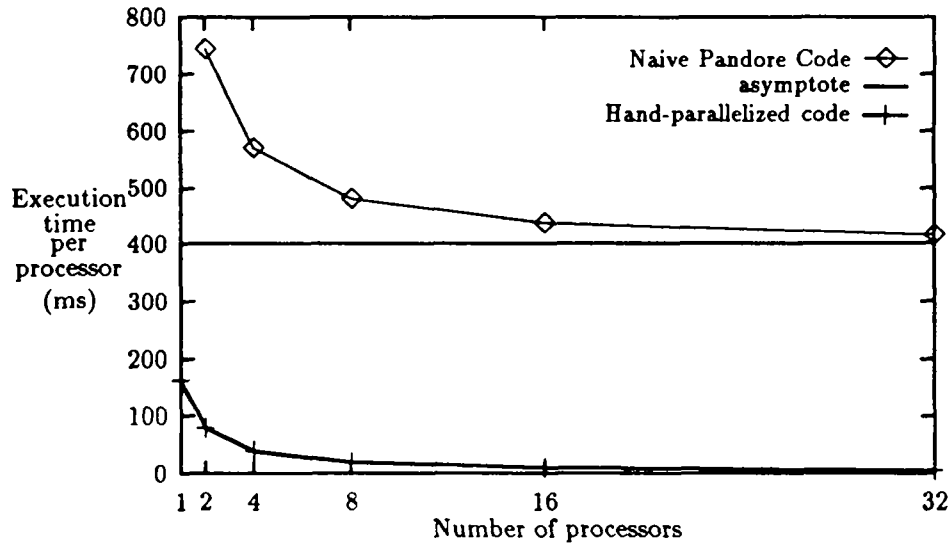

Figure 3: Execution time for vector product: basic PANDORE code vs hand-parallelized code

4.1 Static evaluation of Refresh

The static (at compile-time) evaluation of the **Refresh** statements allows to simplify the code. This evaluation is based on the knowledge of the data decomposition. We perform the following simplification on the code:

```
partition (A, block(T));
partition (B, block(T));
partition (C, block(T));
    ⇒ ∀i, own(A[i]) ≡ own(B[i]) ≡ own(C[i])
for (i=0; i<N; i++)
    Refresh(A[i], own(C[i]));   ⇒ useless statement
    Refresh(B[i], own(C[i]));   ⇒ useless statement
    Exec(C[i]=A[i]*B[i], own(C[i]));
```

The results in execution time of this compile-time optimization is shown figure 4. We see that the execution time has decreased but the bound is still present.

4.2 Static evaluation of Exec

The static evaluation of the **own** part of **Exec** statements enables each processor to execute only the code concerned with its own data. So this will lead to an execution time inversely proportional to the number of processors. If we consider our example, we apply strip-mining to the loop, with respect to the partition bounds:

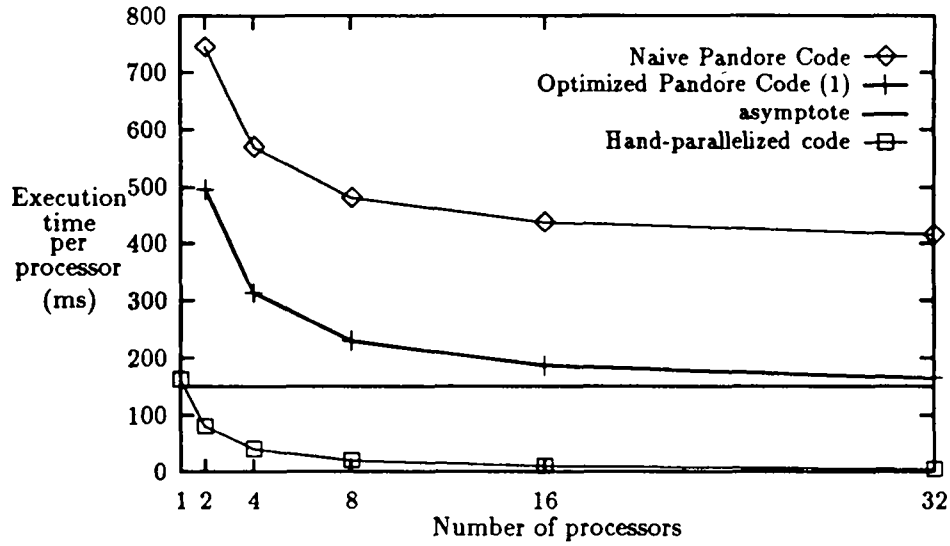

Figure 4: Execution time for vector product: **Refresh** optimized PANDORE code vs hand-parallelized code

```
for (block=0; block< N/T, block++)
    for (i=block*T; i<block*(T+1)-1; i++)
        Exec(C[i]=A[i]*B[i], own(C[i]));
```

We state that each process owns the elements of a block iff it owns one element (the first) of the block.

```
for (block=0; block< N/T, block++)
    Exec(for (i=block*T; i<block*(T+1)-1; i++)
                Exec(C[i]=A[i]*B[i], own(C[i]))
        , own(C[block*T]));
```

We detect that the inner **Exec** is redundant with the external one and we simplify to:

```
for (block=0; block< N/T, block++)
    Exec(for (i=block*T; i<block*(T+1)-1; i++)
                C[i]=A[i]*B[i]);
        , own(C[block*T]));
```

This method can be easily applied to uniform index references ($i + constant$). Only *one* own test per block is performed, and the loop is only executed for local elements. The speedup for this code is shown in figure 5.

We notice that the optimized code exhibits a linear speedup (value $\approx 1/4$). The difference between this speedup and the optimal one (1) is due to the cost of the index transformation

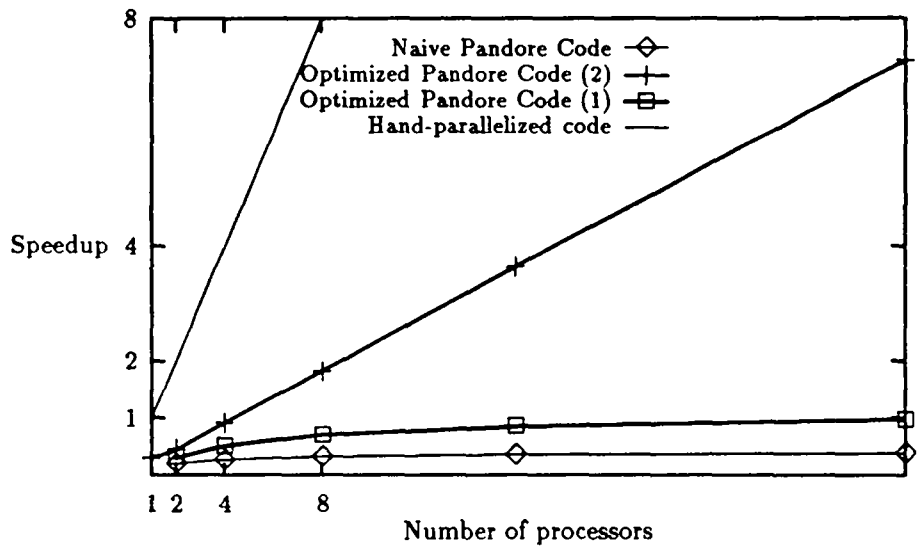

Figure 5: Execution time for vector product: **Refresh** and **Exec** optimized PANDORE code vs hand-parallelized code

performed at compile time, as described at the beginning of this section.

4.3 Communication reorganization

Communications may induce unnecessary synchronisations. We illustrate this fact with the chronogram figure 6, showing the computation of the `X[i]=Y[i-1]` statement in a loop, according that `X` and `Y` are partitioned the same way. In this case, the synchronization sequentializes the computation, although the loop shows no inter-iteration dependence.

In this case, communication reorganization implies to anticipate the emission of `Y[i-1]`. This reorganization can by described by simple transformation rules, essentially permutation of **Refresh** and **Exec** statements. Messages vectorization and use of machine dependent broadcast capabilities can be easily expressed this way. However care must be taken in order:

- not to saturate communication links by anticipating all the emissions at the beginning of the program,
- not to saturate processor memories, by storing too many messages, the transformation scheme being proved only for infinite buffers.

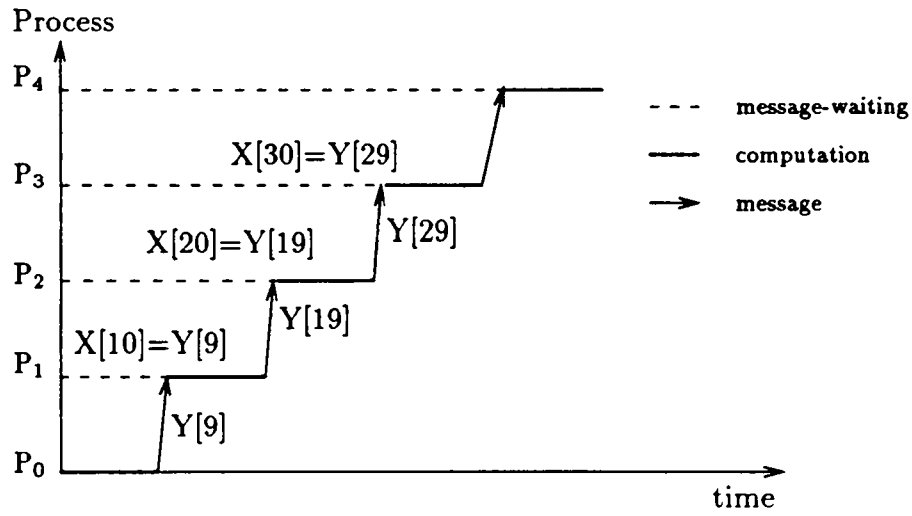

Figure 6: Chronogram for the execution of `for(i=1,i<100;i++) X[i]=Y[i-1]`

5 Conclusion

We have shown that, starting from a pure SPMD model, we may significantly improve the execution performances of a program by performing a static analysis of data and code domains. So, in the PANDORE compiler, the generation of efficient distributed code is mainly related to the partitioning and the mapping of data. Presently, in this system, no help is given to the user for choosing between different partitioning strategies nor for predicting the efficiency of the generated distributed code. Our aim is to provide a static (compile-time) performance evaluator connected to an interactive partitioning system. The informations extracted from the user's data decomposition should allow us to compare the influence of different partitioning schemes on performances. Criterion for performance analysis have to be defined and their accuracy and reliability have to be compared with real execution measurements. We also plan to use run-time performance measurement system in order to tune the compilation rules.

References

[1] M. J. Quinn and P. J. Hatcher. Compiling SIMD programs for MIMD architectures. In *ACM Sigplan (PPEALS)*, 1990.

[2] A. P. Reeves. *The Paragon Programming Paradigm and Distributed Memory Multicomputers.* Technical Report EE-CEG-90-7, Cornell University, June 1990.

[3] M. Rosing, R. B. Schnabel, and R. P. Weaver. *The DINO Parallel Programming Language.* Technical Report CU-CS-457-90, University of Colorado at Boulder, 1990.

[4] C. Koelbel, P. Mehrotra, J. Saltz, and H. Berryman. Parallel loops on distributed machines. In *5' Int. Conf. on Distributed Memory Computing Conference*, April 1990.

[5] E. M. Paalvast and A. J. Van Gemund. A method for parallel program generation with an application to the *Booster* language. In *Int. Conf. on Supercomputing*, pages 457–469, June 1990.

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

[7] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[8] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore : A system to manage Data Distribution. In *International Conference on Supercomputing*, ACM, June 11-15 1990. Egalement publié en Note Interne IRISA.

[9] Henry Thomas. *Une approche de la compilation de programmes séquentiels pour machines à mémoire distribuée.* PhD thesis, IFSIC/Université de Rennes I, june 1991.

LISTE DES DERNIERES PUBLICATIONS PARUES EN 1991

PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER, Claude LE MAIRE
Avril 1991, 36 Pages.

PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED BY SCHEMES
Didier CAUCAL
Avril 1991, 22 Pages.

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Michel RAYNAL
Mai 1991, 24 Pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, ALbert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 Pages.

PI 589 UNE APPROCHE MULTIECHELLE A L'ANALYSE D'IMAGES PAR CHAMPS MARKOVIENS
Patrick PEREZ, Fabrice HEITZ
Juin 1991, 32 pages.

PI 590 THE IDEMPOTENT SOLUTIONS OF THE SEMI-UNIFICATION PROBLEM
Pascal BRISSET, Olivier RIDOUX
Juin 1991, 16 pages.

PI 591 AVARE UN PROGRAMME DE CALCUL DES ASSOCIATIONS ENTRE VARIABLES RELATIONNELLES
Mohamed OUALI ALLAH
Juin 1991, 32 pages.

PI 592 SCHEDULING IN DISTRIBUTED SYSTEMS : SURVEY AND QUESTIONS
Yasmina BELHAMISSI, Maurice JEGADO
Juin 1991, 36 pages.

PI 593 APPLICATION OF BELLEN'S PARALLEL METHOD TO ODE's WITH DISSIPATIVE RIGHT-HAND SIDE
Philippe CHARTIER
Juin 1991, 24 pages.

PI 594 PROGRAMMATION D'UN NOYAU UNIX EN GAMMA
Pascale LE CERTEN, Hector RUIZ BARRADAS
Juillet 1991, 48 pages.

PI 595 CALCULATING THE BUSY PERIOD DISTRIBUTION OF THE M/M/1 QUEUE
Louis-Marie LE NY, Gerardo RUBINO, Bruno SERICOLA
Juillet 1991, 11 pages.

PI 596 EFFICIENT CODE GENERATION FOR DISTRIBUTED MEMORY MACHINES*
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT, Henry THOMAS
Juillet 1991, 14 pages.

ISSN 0249-6399