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

PUBLICATION
INTERNE
N° 869

THE PANDORE COMPILER: OVERVIEW AND EXPERIMENTAL RESULTS

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

The Pandore Compiler: Overview and Experimental Results

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 ° 869 — Octobre 1994 — 24 pages

Abstract: This paper presents an environment for programming distributed memory computers using HPF-like data distribution features. Emphasis is put on compilation techniques and distributed array management. Results are shown for some well known numerical algorithms.

Key-words: Compilation, distributed memory machine, data distribution, loop optimization, parallelization, runtime support.

(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

Le compilateur Pandore : présentation et résultats expérimentaux

Résumé : Ce rapport présente un environnement de programmation des machines à mémoire distribuées utilisant des directives de distribution de données, similaires à celles de HPF. L'accent est mis sur les techniques de compilation et sur la gestion des tableaux distribués. Des résultats expérimentaux sont présentés pour quelques noyaux d'algorithmes numériques.

Mots-clé : Compilation, machine à mémoire distribuée, distribution de données, optimisation de boucles, parallélisation, exécutif.

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 seems to be 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.

Based on this approach, the techniques incorporated in the PANDORE compiler permit generating scalable code. A first technique allows the compilation of any program of the source language but suffers from inefficiencies. In addition, a second method has been defined to optimize parallel loops. An efficient runtime support, that comprises optimized mechanisms for the management of distributed arrays, completes these two schemes.

This paper presents the PANDORE environment and focuses on some optimizations of the compilation scheme and the runtime support. It is organized as follows: the next section briefly presents the PANDORE environment and section 2.2 gives an overview of the PANDORE language. 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

2.1 Overview

We have developed an experimental programming environment for DMPC. The kernel of this environment is the PANDORE compiler, built according to the basic principles discussed above.

When we designed the compiler, our objectives were the following:

- *to build a modular and extensible compiler:*
  In order to be able to integrate new techniques easily according to the state of art, we rely on the specification of an intermediate program representation which does not depend on the input language and on which all analysis and transformation techniques may take place. The compiler has been written using the CAML language [ALM*90] which is a functional language based on the ML language providing polymorphic type synthesis, pattern matching and facilities to define parsing functions. Moreover, we have defined an abstract machine level to achieve the generation of machine independent code.

- *to use general and safe techniques:*
  Our aim is to distribute and parallelize any program which fulfills the chosen input language syntax. Thus, we rely on an implementation of the owner-writes rule that has been proved to be correct [BCJT92]. Moreover, by a suitable runtime system and sophisticated compilation techniques, we obtain efficient execution for a reasonably large class of programs.

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.

A translator from HPF to the PANDORE language is also provided [JAC*94]. HPF includes some "standard" extensions for data parallelism and data distribution in the Fortran 90 language. It has been designed by the High Performance Fortran Forum from March 1992 to May 1993. HPF permits to distribute arrays among virtual processor arrays (TEMPLATES); see [For93] for a complete description of the language.

The PANDORE run-time uses a generic message passing library called POM (Parallel Observable Machine [GM94]). This library offers limited but efficient services. It allows to run the same program on a wide range of distributed memory computers.

2.2 The Pandore Language

The PANDORE language is based on a sequential imperative language using a subset of C (excluding pointers) as a basis. We have added a small set of simple data distribution features in order to describe frequently used decompositions. See [Che93] for a more precise definition of the language. If other regular decomposition functions are seen to be necessary, they will be added to the language.

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 specification of the partitioning for a d-dimensional array is given by the keyword **block** $(t_1, ..., t_d)$ where t_i indicates the size of the blocks in the i^{th} dimension. This is similar to the HPF `CYCLIC(`k`)` distribution [For93] except for the mapping. For example

$$Y[NC][MC] \textbf{ by block } (1, MC)$$

indicates that the array Y of $NC \times MC$ elements is decomposed into blocks of size $1 \times MC$: the array is decomposed into NC lines.

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.

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.

2.2.2 Other Features

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. Closed functions are similar to HPF PURE functions.

```
#define N 128

dist LU(
 double A[N][N] by block(1,N) map wrapped(1,0) mode INOUT)
{
 int i,j,k;

 for (k=0; k<(N-1); k++) {
   for (i=k+1; i<N; i++)
      A[i][k] = A[i][k] / A[k][k];
   for (i=k+1; i<N; i++)
      for (j=k+1; j<N; j++)
         A[i][j] = A[i][j] - A[i][k] * A[k][j];
 }
}
```

Figure 2: Kernel of the LU factorization algorithm

2.3 Related Work

Many other research projects are addressing the definition of programming environments for running data parallel programs on distributed MIMD computers. Among them, the Vienna Fortran Compilation System (VFCS) and the Fortran D system are well known. Preliminary research related to these projects and to the PANDORE project began in 1988, hence Fortran D, Vienna Fortran and the PANDORE language have been defined anteriorly to HPF.

VFCS is based on the Superb tool [ZBG88] that has been designed by M. Gerndt and H. Zima. Superb is a semi-automatic tool; data distribution is interactively specified and not included in the source language. The Vienna Fortran language has been defined later on [ZC92]. It includes data distribution features and alignements but does not use templates, as opposed to HPF. VFCS is a source to source translator; it implements the overlap concept and some support for irregular computations.

Fortran D has been initiated by K. Kennedy who laid the foundation of the owner-writes rule and the run-time resolution technique. In the Fortran D language, arrays are aligned with virtual index spaces called decompositions that are distributed on the processors. This language is relatively close to HPF. The compiler performs some optimizations like loop bounds reduction and message vectorization [Tse93].

These three environments are still prototype environments, none of them can handle efficiently a very large set of applications, each of them solving different compilation problems. Researchers from the VFCS and PANDORE groups are working within the Prepare European Esprit Project whose aims is to build a HPF programming environment of industrial quality [AM93].

3 The Pandore Compiler

The compiler uses two techniques to generate SPMD code from the user-supplied data decomposition with respect to the owner-writes rule. The first one is known as *runtime resolution* [CK88] and is employed in most compilers for HPF-like languages; this technique does not require complicated analysis from the compiler and can be used with any input program to produce a distributed code preserving the dependences of the input program. For parallel loops or reductions that can be analyzed at compile time, a more sophisticated technique is used to generate efficient SPMD communication and computation codes.

3.1 The Runtime Resolution Technique

The application of this basic scheme requires a simple analysis from the compiler. Roughly speaking, the technique consists in transforming each assignment of the program: for each right hand side (*rhs*) reference to a distributed array, the compiler generates the appropriate *send* and *receive* statements to ensure cooperation between the processors owning respectively the *rhs* and the left hand side (*lhs*) reference. Then, the compiler inserts a mask so that the assignment is performed only on the processor owning the *lhs* reference. Figure 3 shows an example of a simple input program with the corresponding pseudo-code produced with the runtime resolution technique.

It is well known that this technique yields inefficient code. As it can be seen on figure 3, communications are performed elementwise and the entire iteration domain associated with the input loop is scanned by all the processors: for each iteration step, each processor evaluates masks to discover if it has to perform a communication, an assignment, or nothing.

```
                                  for i = 1 , 100
                                      for j = 1 , 100
for i = 1 , 100                           if myself ≠ owner(A[i,j]) ∧ myself = owner(B[j,i])
    for j = 1 , 100          ⇒                then send B[j,i] to owner(A[i,j])
        A[i,j] := B[j,i]                  if myself = owner(A[i,j]) ∧ myself ≠ owner(B[j,i])
                                              then receive B[j,i] from owner(B[j,i])
                                          if myself = owner(A[i,j]) then A[i,j] := B[j,i]
```

Figure 3: Runtime resolution

3.2 Compiling Parallel Loops and Reductions

Framework

The compiler optimization effort concentrates on reductions and parallel loops with one statement. This kind of loop is frequently used in scientific programs; furthermore, when they are not explicitly present, these loops can be produced thanks to automatic parallelization techniques. For example, affine-by-statement scheduling [DR92, Dar92] can be applied on loop nests with uniform or affine dependences and produces a sequential loop indexed by time, whose body is composed of parallel loops with one statement.

As the optimized method we have developed is based on the *polyhedron* model, we suppose that array references and loop bounds are affine functions of the enclosing indices. Concerning data distribution, we assume that arrays are partitioned into blocks with constant size; the optimized scheme is described when each block is owned by only one processor but the technique can be trivially extended to handle block replication. Because the mapping of the blocks is not taken into account during the optimization process, every mapping is supported for the blocks of an array. Alignment is not available in the PANDORE language so this feature is not presently treated. Finally, it should be noted that the analysis carried out in the compiler is symbolic, that is to say independent of the number of blocks of the arrays involved in the parallel loop or the reduction.

In this rather general framework, our optimized compilation scheme embeds several optimizations such as restriction of iteration domains and message vectorization.

Related Work

Other existing compilers for HPF-like languages often target a larger application domain than PANDORE presently does (such as programs with irregular loops) and thus use various translation and execution techniques. However, concerning the optimization of regular parallel loops, that is in a framework comparable to the one described above, the compile-time analysis they perform at present can be applied on a more restricted class of loop bounds, access patterns and distributions. In the Vienna Fortran Compilation System (VFCS) [BCZ92], the loop bounds and the array access functions are affine functions of one enclosing index. In the Fortran D compiler, the restrictions are the following: the loop bounds must be constant, the array access functions are of the form `i+c` (`i` is a loop index and `c` a constant known at compile-time) and only one dimension of an array can be distributed. For both compilers, arrays involved in the parallel loop must be mapped in such a way that each processor only owns one block of an array, hence they can not handle general cyclic distribution. Furthermore, both compilers perform a domain analysis for each processor, leading to a compilation time that depends on the number of processors.

In VFCS, communication code generation and communication optimization (as well as storage allocation) rely on an *overlap analysis*. For each processor and for each array appearing in the *rhs*, the overlap analysis aims at approximating the union of the set of local elements and the set of distant elements needed by a rectangular section (*overlap area*). Because the compilation technique relies on the SPMD model, the overlap area for an array is defined as the smallest rectangular section that can contain all the overlap areas computed for each processor. Communication code generation and communication optimization for an array are then performed, depending on the analysis of the intersections between the overlap area and the distant blocks. For the generation of the computation code, the compiler performs a symbolic loop bounds reduction: the new loop bounds depends on the processor identity.

The Fortran D compiler performs domain analysis using rectangular-triangular sections called *regular sections*. For each *rhs* reference $A[f(I)]$ and for each pair of processors (p, p'), the compiler carries out communication code generation and communication optimization thanks to the computation of two index sets: $IN(A[f(I)], p, p')$ and $OUT(A[f(I)], p, p')$. These sets define the indices J of A such that p must receive $A[J]$ from p' (IN) and p must send $A[J]$ to p' (OUT). Regarding computation code generation, the compiler performs loop bounds reduction for each processor p. This reduction is based on the computation of the iteration set associa-

ted with the *lhs* reference $B[g(I)]$, that is, the set of iterations that cause $B[g(I)]$ to access data owned by p.

Other optimization techniques focusing on parallel loop optimization and using the polyhedron model are also being investigated [IACK93] but are not implemented yet.

A different approach, that is neither based on parallel loop optimization nor on the owner-writes rule, is set out in [AL93]. This technique handles loop nests with affine loop bounds and access functions and takes as input a data decomposition and a computation decomposition. It characterizes SPMD communication sets (using *data-flow analysis*) and the SPMD computation set with polyhedrons, computes the loops that enumerate these sets and then performs *loop splitting* to produce the final target code. This technique has not been fully integrated in a compiler and seems to be applicable only if the compiler can effectively handle a lot of possibly complex polyhedrons and if loop splitting does not lead to an unacceptable target code fragmentation.

Optimization Process

A complete description of the compilation scheme of PANDORE can be found in [LPA93]. In this paper, we only describe our approach through the example of the LU factorization algorithm given in figure 2.

First, reductions and parallel loops with one statement need to be identified (as dependence tests are not yet included in the compiler, we rely on user annotations). In the LU factorization, the k-loop is not parallel so the iteration domain associated is replicated on all the processors. On the other hand, the inner i-loop and (i,j)-loop, parameterized by k, are parallel. Thus, the compiler applies the optimized compilation scheme for both of them. The remaining of this section details the code generation for the (i,j)-loop. This code comprises a communication part and a computation part.

Communication Code Generation

The compiler produces a sequence of communication codes, one communication code being defined for each *rhs* reference to a distributed array in the (i,j)-loop assignment. Each communication code is composed of a send part and a dual receive part. For sake of brevity, we will focus on the production of the communication code related to the $A[k,j]$ reference. Codes for the other references can be then easily figured out.

First, subscripts and array partitioning analysis is performed for *lhs* reference $A[i,j]$ and *rhs* reference $A[k,j]$ so that the compiler constructs the system of affine constraints:

$$\begin{cases} 0 \leq kAl \leq 127 \\ 0 \leq kAr \leq 127 \\ k+1 \leq i \leq 127 \\ k+1 \leq j \leq 127 \\ u = k \\ v = j \\ kAl \leq i \leq kAl \\ kAr \leq u \leq kAr \end{cases}$$

which defines the set of vectors (kAl, kAr, u, v, i, j) in which the array element $A[u,v]$, located in the block number kAr of A, is needed to perform the writings on the the block number kAl of A (array A is partitioned into 128 lines numbered from 0 to 127). The above system of constraints defines a polyhedron P parameterized by k which is then projected along the i, j axis; this results in a new polyhedron P_{ij} whose enumeration code is computed by the method described in [LeF94]. This yields the nested loop:

for $kAl = k+1\ ,\ 127$
  for $kAr = k\ ,\ k$
    for $u = kAr\ ,\ kAr$
      for $v = k+1\ ,\ 127$

From this loop, the compiler generates the send code and the dual receive for reference $A[k,j]$. 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 runtime:

for $kAl = k+1\ ,\ 127$
  if *myself* $\neq$ owner_block($A,\ kAl$)
    for $kAr = k\ ,\ k$
      if *myself* = owner_block($A,\ kAr$)
        for $u = kAr\ ,\ kAr$
          RLR_send $\{A[u,v]/k+1 \leq v \leq 127\}$ to owner_block($A,\ kAl$)

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 runtime resolution. Furthermore the (kAl, kAr)-loop do not scan the whole cartesian product

$0..127 \times 0..127$ and the location of the first mask prevents from enumerating all the vectors described by the (kAl, kAr)-loop.

From the description of the elements $A[u, v]$ to be sent, the runtime 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, 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.
- Elimination of redundant communications is performed when several references to the same distributed array appears in the right hand side.

Computation Code Generation

The SPMD computation code is generated as follows: the compiler analyzes subscripts and array partitioning for the *lhs* reference $A[i, j]$ to build the set of constraints:

$$\begin{cases} 0 \leq kA \leq 127 \\ k+1 \leq i \leq 127 \\ k+1 \leq j \leq 127 \\ kA \leq i \leq kA \end{cases}$$

which defines the vectors (kA, i, j) where iteration vector (i, j) is such that *lhs* reference $A[i, j]$ writes in block kA of A. The enumeration code for the polyhedron associated with the previous system is then computed using the same techniques as in the communication code generation:

for $kA = k + 1\ ,\ 127$
  for $i = kA\ ,\ kA$
    for $j = k + 1\ ,\ 127$

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

```
for kA = k + 1 , 127
    if myself = owner_block(A, kA)
        for i = kA , kA
            for j = k + 1 , 127
                A[i,j] = A[i,j] − A[i,k] ∗ A[k,j]
```

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 runtime is introduced at the block level and second, the outer k_A-loop does not scan the whole interval 0..127.

4 Management of Distributed Arrays

4.1 Rationale

Representation of distributed arrays as well as accesses to elements of these arrays is a critical issue for overall performance of the produced code. Any sophisticated compilation scheme is useless if no effort is done on the management of distributed arrays in terms of time efficiency and memory requirement.

We have to achieve a trade-off between the speed of accesses and the memory overhead induced from the array representation. The extreme solution consisting in allocating the entire array on each processor is obviously not applicable. Conversely, minimal allocation associated with index conversion involving several costly operations such as *mod* and *div* has to be avoided.

For a given processor p, accesses to array elements can be divided in two categories: accesses to local elements – i.e. elements assigned to p by the distribution – and accesses to elements previously received from other processors. Our optimized compilation scheme generates code in which communication is separated from computation and no difference is made in the computation code between accesses to local and received elements. Therefore, if a non-uniform representation had been used (for instance by allocating a sub-array for local data and managing buffers and hash-tables for received data), an ownership computation would have been required at runtime for each access in order to use the appropriate access mechanism, reducing the benefit of the optimizations made by the compiler. It should be added that even if such a separation is possible at compile-time, it may induce an important code fragmentation.

We want to provide an array management scheme defined only from the distribution parameters (not from the code itself) and thus consider that only global indices appear in the generated code. This independence facilitates the use of different compilation techniques within a code fragment that contains several loops. Moreover, an array management scheme that depends on the analysis of the loops and access patterns might have led to different layouts and access mechanisms within the scope of one distribution. In this case, data rearrangement or additional computation would have been needed at runtime between loops to switch from an array management scheme to another.

Another useful property concerning the layout of distributed arrays is the conservation of memory contiguity. Indeed, if contiguous elements of the original array are still contiguous in the local representation, it makes it possible to take advantage of direct communications, target code optimization and better cache behavior.

We have adopted a management of distributed data structures based on the paging of arrays. This is a general and uniform management of local and received data. It has been designed in order to achieve efficient accesses while avoiding unacceptable memory overhead.

4.2 Related Work

To our knowledge, management of distributed arrays have not been studied independently from compilation techniques in existing HPF-like languages compilers. The first technique of storage for distributed arrays, *the overlap* [ZBG88] has been implemented in the Vienna Fortran Compilation System [ZC92] and in the Fortran D compiler [Tse93]: a single sub-array is allocated for local data as well as for received data. This technique provides uniform and efficient accesses but can be applied to a restricted number of distributions and access patterns and may lead to the replication of the whole array. The Fortran D compiler may select one of two alternative storage methods for received values (buffers and hash tables) when it can separate purely local computation and computation needing received values.

Although they have not been integrated in complete prototypes, other techniques have been proposed. In the compilation scheme defined by Ancourt *et al* [IACK93], local elements and temporaries are packed according to the array distribution and alignment, the loop bounds and the array subscripts by changing the basis of the original index space. Accesses to elements are performed in a non-uniform way with index conversion by evaluating affine functions and possibly integer division. Chatterjee *et al.* [CGST93] propose an access mechanism for local elements based on a Finite State Machine (FSM). These elements are accessed by executing a FSM

that has to be computed at runtime for each loop nest even if the same distribution applies.

All of these methods not only take into account the array distribution parameters but necessitate also a static analysis of code fragments (loop bounds and array subscripts) in order to define the layouts of the local arrays and the associated access mechanisms.

4.3 Paging Distributed Arrays

In PANDORE, arrays are managed by a *software* paging system. The runtime 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 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 runtime. One of the advantages of paging arrays is that accesses to local and received elements are performed the same way. Indeed, as far as accesses are concerned, a processor acts as if the entire array was directly visible, no matter if the element it needs to access is local or has been received from another processor.

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 [MP93].

4.4 Efficiency of the Paging System

As far as speed of access is concerned, paging of distributed arrays gives very satisfactory results: times remain very close to times for accesses without index transformations. Table 1 shows the results of a preliminary experiment. We considered the assignment to a scalar and measured the time taken by this assignment for several right-hand-sides:

- t_c : *rhs* is a literal constant;
- t_s : *rhs* is a reference to an element as it may appear in a sequential program;
- t_p : *rhs* is a call to the macro that uses the paged access mechanism;
- t_b : *rhs* is a call to a macro that uses a block-oriented access mechanism. This mechanism performs a modulo and an integer division at runtime to find the block number and the offset in the block.

The array is a two-dimensional array of floats. Reported times (in μs) are the differences $t_s - t_c$, noted *sequential*; $t_p - t_c$, noted *page* and $t_b - t_c$, noted *block*. Best and worst cases have been considered, depending on whether sizes of the array were powers of two or not. Experiments have been carried out on a SparcStation 2, on a node of the Intel iPSC/2 and on a node of the Intel Paragon XP/S. Native compilers have been used with no optimization option.

	Sparcstation		iPSC/2		Paragon	
	best	*worst*	*best*	*worst*	*best*	*worst*
Sequential	0.30	0.42	0.94	2.05	0.16	0.26
Page	0.34	0.38	2.14	2.26	0.22	0.25
Block	0.48	1.58	3.52	9.86	0.21	2.68

Table 1: Time for accessing array elements (in μs).

The memory overhead induced by paging is almost entirely due to the tables of pages. Indeed, when a page contains elements that have no equivalent in the original

array space, or when just a part of a distant page is accessed in an optimized loop, only a portion of the page is actually allocated.

Table 2 gives memory requirements for a few common distributions of arrays on 32 processors. For each distribution, we indicate the total number of pages, the theoretical minimal memory space required on each processor, the actual space allocated for tables on each processor and finally the overhead as compared with the minimal partition. Memory needs are expressed in bytes. It can be noticed that replacing some block sizes (or array dimensions) by powers of two notably decreases the memory overhead. We believe that overall memory requirements remain acceptable when considering most commonly used distributions.

Array Distribution	*Number of Pages*	*Minimal Partition*	*Local Space for Tables*	*Local Overhead*
double A[100000] by block(1000)	196	25000	1960	8%
double A[100000] by block(1024)	98	25000	588	2%
double A[1000][1000] by block(1,1000)	1000	250000	6000	2%
double A[1000][2000] by block(50,500)	8000	500000	80000	16%
double A[1000][2000] by block(50,512)	4000	500000	24000	5%
double A[100][100][100] by block(100,1,50)	10000	250000	60000	24%

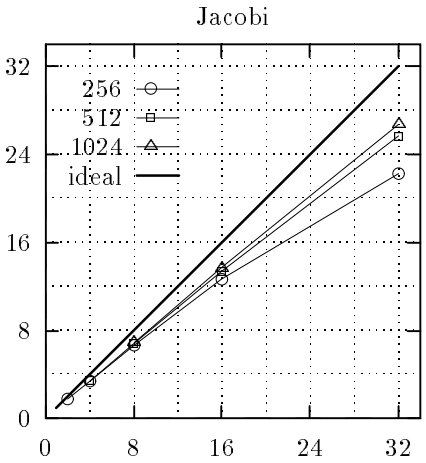
Table 2: Memory requirements for a few common distributions

5 Experimental Results

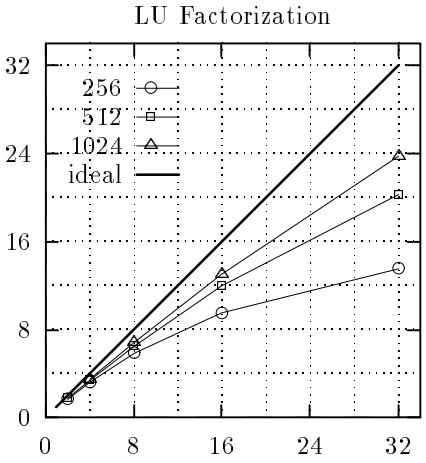
Some results of experiments with the optimized compilation scheme and the paged array management are presented in this section. The compilation of several well-known kernels have been tested with the PANDORE environment; the parallelization of a wave propagation application is set out in [ALMP94]. The kernels studied here are the following: Jacobi relaxation, Red-Black SOR, Daxpy, Matrix-matrix product, Cholesky and LU factorizations, Modified Gram-Schmidt algorithm. The source code of the distributed phases of PANDORE programs 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 communication.

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. Sequential times for

data sizes that could not fit in the memory of one node have been estimated. The obtained efficiencies are satisfactory ranging from 80% to 95% on 8 processors and from 75% to 85% on 32 processors for the largest data size although the ratio of memory operations to computation is often high.

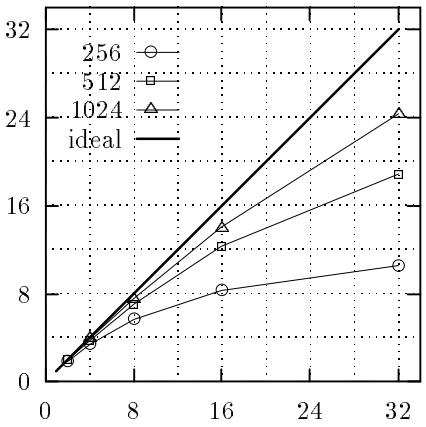



```
dist jacobi(double B[N][N] by block(N/P,N)
            map regular(0,1) mode INOUT)
double A[N][N] by block(N/P,N) map regular(0,1);
{
 int i,j;

 for (i=1; i<N-1; i++)
  for (j=1; j<N-1; j++)
    A[i][j] = V*B[i][j] +
      W*(B[i-1][j]+B[i+1][j]+B[i][j-1]+B[i][j+1]);

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



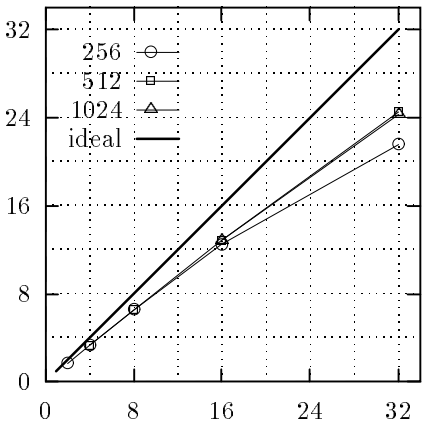

```
dist LU(double A[N][N] by block(1,N)
        map wrapped(1,0) mode INOUT)
{
 int i,j,k;
 double line_k[N], corner;

 for (k=0; k<(N-1); k++) {
   corner = A[k][k];
   for (i=k+1; i<N; i++)
      A[i][k] = A[i][k] / corner;
   for (j=k+1; j<N; j++)
      line_k[j] = A[k][j];
   for (i=k+1; i<N; i++)
      for (j=k+1; j<N; j++)
         A[i][j] = A[i][j] - A[i][k] * line_k[j];
 }
}
```

Cholesky Factorization

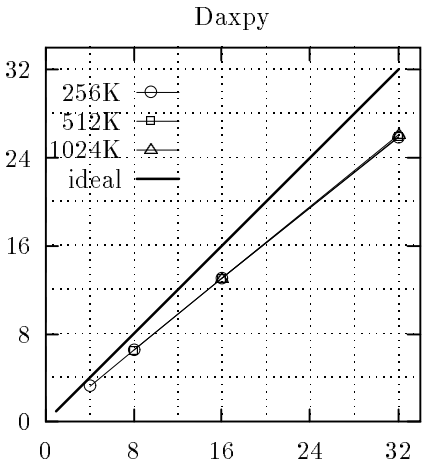



```
dist cholesky(double A[N][N] by block(N,1)
              map wrapped (0,1) mode INOUT)
{
 int i,j,k;
 double colk[N];

 for(k=0; k<N; k++) {
    A[k][k] = sqrt(A[k][k]);
    for(j=k+1; j<N; j++)
      A[j][k] = A[j][k] / A[k][k];
    for (j=k+1; j<N; j++)
       colk[j] = A[j][k];
    for(j=k+1; j<N; j++)
      for(i=j; i<N; i++)
        A[i][j] = A[i][j] - colk[i] * colk[j];
 }
}
```

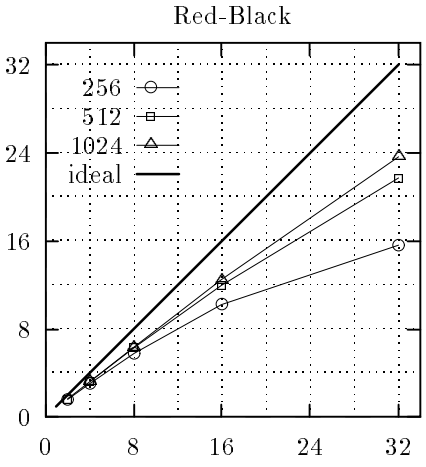
Matrix-matrix product



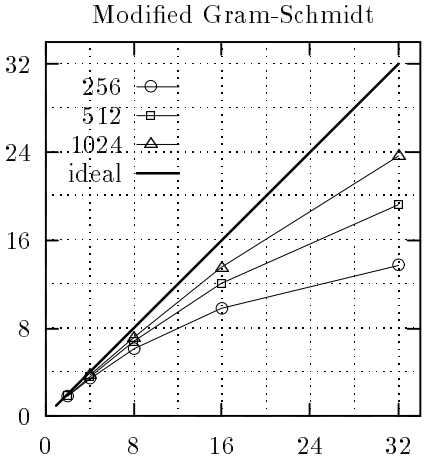

```
dist matprod(double A[N][N] by block(N/P,N)
                map regular(0,1) mode IN,
             double B[N][N] by block(N,N/P)
                map regular(0,1) mode IN,
             double C[N][N] by block(N/P,N)
                map regular(0,1) mode OUT
            )
{
 int i,j,k;
 double colj[N];

 for (i=0; i<N; i++)
    for (j=0; j<N; j++)
       C[i][j] = 0.0;
 for (j=0; j<N; j++) {
    for (k=0; k<N; k++)
       colj[k] = B[k][j];
    for (i=0; i<N; i++)
       for (k=0; k<N; k++)
         C[i][j] = C[i][j] + A[i][k]*colj[k];
 }
}
```




```
dist daxpy(
 double alpha mode IN,
 double X[N] by block(N/P) map regular(0) mode IN,
 double Y[N] by block(N/P) map regular(0) mode INOUT)
{
 int i;

 for (i=0; i<N; i++)
    Y[i] = alpha * X[i] + Y[i];
}
```




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

 for (i=0; i<(N-1)/2; i++)
    for (j=0; j<(N-1)/2; j++)
       A[2*i+1][2*j+1] =
          (W/4) * (A[2*i][2*j+1] + A[2*i+2][2*j+1] +
                   A[2*i+1][2*j] + A[2*i+1][2*j+2])
                + A[2*i+1][2*j+1] * (1-W);
  for (i=1; i<(N-1)/2+1; i++)
    for (j=1; j<(N-1)/2+1; j++)
       A[2*i][2*j] =
          (W/4) * (A[2*i-1][2*j] + A[2*i+1][2*j] +
                   A[2*i][2*j-1] + A[2*i][2*j+1])
                + A[2*i][2*j] * (1-W);
   for (i=1; i<(N-1)/2+1; i++)
    for (j=0; j<(N-1)/2; j++)
       A[2*i][2*j+1] =
          (W/4) * (A[2*i-1][2*j+1] + A[2*i+1][2*j+1] +
                   A[2*i][2*j] + A[2*i][2*j+2])
                + A[2*i][2*j+1] * (1-W);
   for (i=0; i<(N-1)/2; i++)
    for (j=1; j<(N-1)/2+1; j++)
       A[2*i+1][2*j] =
          (W/4) * (A[2*i][2*j] + A[2*i+2][2*j] +
                   A[2*i+1][2*j-1] + A[2*i+1][2*j+1])
                + A[2*i+1][2*j] * (1-W);
}
```




```
dist MGS(double v[N][N] by block(1,N)
         map wrapped(0,1) mode INOUT)
     double xnorm[N] by block(1) map wrapped(0);
     double sdot[N]  by block(1) map wrapped(0);
{
  int i,j,k;
  double vc[N];

  for (i=0;i<N;i++) {
    xnorm[i] = 0.0;
    for (k=0;k<N;k++)
      xnorm[i] = xnorm[i] + v[i][k]*v[i][k];
    xnorm[i] = 1.0/sqrt(xnorm[i]);
    for (k=0;k<N;k++)
      v[i][k]=v[i][k]*xnorm[i];
    for (k=0;k<N;k++)
      vc[k]=v[i][k];
    for (j=i+1;j<N;j++) {
      sdot[j]=0.0;
      for (k=0;k<N;k++)
        sdot[j]=sdot[j]+vc[k]*v[j][k];
      for (k=0;k<N;k++)
        v[j][k]=v[j][k] - sdot[j]*vc[k];
    }
  }
}
```

6 Conclusion

The two compilation techniques described in this paper have been fully implemented in the compiler and permit the compilation of the whole PANDORE language. They benefit from the efficiency of our page-driven array management scheme. The performances obtained on a series of numerical kernels are already satisfactory even though enhancements can be made along several axes.

We plan to improve the compiling scheme in order to produce a more efficient code that overlaps communication and computation. Also, the use of threads is studied that should avoid undue sequentialization of communication operations within the generated code.

In order to compile irregular data accesses efficiently, we are investigating the integration of the inspector/executor technique [DSB92] as well as the exploitation of shared virtual memory. The next step will be combining data-parallelism and control-parallelism homogeneously in the PANDORE environment.

References

[AL93] 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.

[ALM*90] M.V. Aponte, A. Lavaille, M. Mauny, A. Suarez, and P. Weis. *The CAML Reference Manual.* Technical Report, INRIA, 1990.

[ALMP94] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. *Parallelization of a Wave Propagation Application using a Data Parallel Compiler.* Technical Report 868, IRISA, October 1994.

[AM93] A. Veen and M. de Lange. Overview of the prepare project. In *4th International Workshop on Compilers for Parallel Computers, Delft*, pages 345–350, December 1993.

[BCJT92] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. *Correctness of Automated Distribution of Sequential Programs.* Research Report 665, IRISA, June 1992.

[BCZ92] 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.

[CGST93] 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.

[Che93] O. Chéron. *Pandore II : un compilateur dirigé par la distribution des données.* PhD thesis, IFSIC/Université de Rennes I, July 1993.

[CK88] D. Calahan and K. Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, October 1988.

[Dar92] A. Darte. *Affine-by-statement Scheduling: Extensions for Affine Dependences and Several Parameters.* Research Report 92-03, LIP, Lyon, France, June 1992.

[DR92] A. Darte and Y. Robert. *Affine-by-statement Scheduling of Uniform Loop Nests over Parametric Domains.* Research Report 92-16, LIP, Lyon, France, April 1992.

[DSB92] R. Das, J. Saltz, and H. Berryman. *A Manual for PARTI Runtime Primitives - Revision 1.* Technical Report, ICASE, Langley Reseach Center, Hampton VA 23065, December 1992.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[GM94] F. Guidec and Y. Mahéo. *POM: a Parallel Observable Machine.* Technical Report, IRISA, 1994. To appear.

[IACK93] 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.

[JAC*94] L. Jerid, F. André, O. Chéron, J.-L. Pazat, and T. Ernst. *HPF to C-Pandore Translator.* Technical Report 2283, INRIA, May 1994.

[LeF94] M. Le Fur. *Parcours de polyèdre paramétré avec l'élimination de Fourier-Motzkin.* Technical Report 858, IRISA, Rennes, France, September 1994.

[LPA93] M. Le Fur, J.L. Pazat, and F. André. *Static Domain Analysis for Compiling Commutative Loop Nests.* Technical Report 2067, INRIA, France, October 1993.

[MP93] Y. Mahéo and J.-L. Pazat. *Distributed Array Management for HPF Compilers.* Research Report 2156, INRIA, December 1993.

[Tse93] C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[ZBG88] H. P. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, (6):1–18, 1988.

[ZC92] 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.