Distributed array management for HPF compiler

Yves Mahéo, Jean-Louis Pazat

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Distributed Array Management for HPF Compilers

Yves Mahéo, Jean-Louis Pazat

N° 2156

Décembre 1993

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Distributed Array Management for HPF Compilers

Yves Mahéo, Jean-Louis Pazat*

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

Rapport de recherche n°2156 — Décembre 1993 — 14 pages

Abstract: This paper addresses the management of distributed arrays for HPF compilers. We present an efficient method to allocate local blocks and temporaries for received values and manage the associated access mechanisms. The performance of these access mechanisms is measured and experimental results on the use of this array management within a compiler are shown.

Key-words: Distributed memory parallel computers, compilation, HPF, runtime, paging, memory management

(Résumé : tsvp)

*maheo@irisa.fr, pazat@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Gestion des tableaux distribués pour les compilateurs HPF

Résumé : Ce rapport traite de la gestion des tableaux distribués pour les compilateurs HPF. Nous présentons une méthode pour gérer efficacement l'allocation et les accès concernant les blocs locaux et les temporaires de réception. L'efficacité du mécanisme d'accès est mesurée et des résultats expérimentaux sur l'utilisation de cette gestion des tableaux dans un compilateur sont donnés.

Mots-clé : Machines parallèles à mémoire distribuée, compilation, HPF, exécutif, pagination, gestion mémoire

1 Introduction

In order to alleviate the task of programming Distributed Memory Parallel Computers, new features have been added to sequential programming languages such as Fortran. In the field of scientific programming, two main axes are currently followed. The first one uses explicit parallel constructs (DOALL) and relies on a shared virtual memory [3]; the second one is based on a user-defined data distribution which is used as a guideline to generate communicating processes [8].

In recent years, many projects have focused on the *data distribution* approach and it has been demonstrated that "aggressive" optimizing compilers and efficient runtime systems are mandatory to achieve reasonable speedups. Most compilers allow the user to specify a decomposition of arrays and use the *owner write rule* [5] to distribute the code. This distribution can be done using the runtime resolution technique where each statement of the source program is guarded and where communication is performed elementwise. This scheme is always applicable but rather inefficient, so that many compilers integrate optimization techniques for compiling loops. Roughly speaking, these techniques aim at reducing iteration domains and performing vectorized communications [13, 11, 10].

However, runtime resolution as well as optimization techniques require a specific and efficient runtime system to allocate local parts of arrays and perform efficient communications.

In this paper we present a method for efficiently managing distributed arrays (allocation and access) in parallelizing compilers based on data distribution. This paper focuses on the local management of blocks and temporary storage for distant values. Communication optimizations such as message coalescing and vectorization are supported by this array management but are not addressed here.

The paper is organized as follows: next section discusses the essential requirements for the runtime system. Section 3 details the page-driven array management proposed. Section 4 presents in more details the implementation of this array management whose overall performance is presented in section 5. Future work is discussed in the conclusion.

2 Requirements of Distributed Array Managements

2.1 Memory Management

Most compilation systems for HPF-like languages used to –or still– adopt more or less the same method for representing distributed arrays and, as a consequence, for accessing elements of these arrays. Processors are allocated minimal space for local partitions (i.e. local blocks), reflecting or not the multidimensionality of the original array [13, 9, 4]. Typically, the declaration of an array $A[N][N]$ distributed in every dimension will be translated into a local declaration $A[N/P][N/P]$ where P is the number of processors. The memory overhead induced by this layout remains small. Proposals have also been made to allow for alignment while keeping memory use at a minimum [6]. To complement the management of local partitions, several *ad hoc* techniques (temporary scalars [2], buffers [11], hash tables [7], extension of local partition [12]) are used for handling received data. The additional memory space required varies according to the chosen technique but must remain acceptable.

2.2 Global to Local Index Conversion

In order to take into account a large class of programs, compilers must be able to produce code where indices remain global. Thus accesses to local elements are performed thanks to index transformation functions that convert global indices of the source program into indices adapted to the local layout of the partition. These functions take into account the decomposition of the array into blocks and possibly its alignment with a template and its mapping onto the processors. Some compilers can sometimes avoid global to local index conversion at runtime, but it should be hightlighted that this is only possible for some particular distributions (essentially block distributions) and for some particular loops and access patterns. The computation of conversion functions is costly because it generally involves several operations such as modulo and integer division; it may induce a ratio of 10 as compared with an access without conversion. For accessing nonlocal elements, as for accessing local elements, some kind of index conversion is likely to be needed at runtime. Its cost depends on the technique used but may also be rather high.

2.3 Uniform Representations and Accesses

The distributed array managements described above not only bring about possibly costly index transformations, but they may also pose a problem of uniformity if they store and access local and received data differently.

To illustrate how this problem arises, let us consider the example of an earlier version of the PANDORE environment. For each distributed array, local partitions were composed of linearized local blocks, so that global to local index conversions had to be performed. The application of an optimized compilation scheme for parallel loops permitted the separation of the SPMD code produced into a communication phase and a computation phase [10]. Elements received during the communication phase were stored in hash tables. For the computation phase, no difference was made by the compiler between accesses to local and received elements. Thus an ownership computation, comprising an evaluation of the block number, was performed for each access in order to choose the appropriate global to local index conversion function. Consequently, the benefit of the optimizations made by the compiler was reduced due to the non-uniform accesses to data.

In other words, a non-uniform management induces a loss of performances if the compiler cannot separate local accesses from accesses to received elements. Besides, even if compile-time separation can be achieved, it may yield an unacceptable code fragmentation in the case of multiple right-hand-side references.

Several uniform managements are yet known. The first one consists in the replication of the array: it allocates the entire array on each processor in order to access elements the same way as in the sequential program. It is obvious that the price to pay for uniformity in terms of memory use is not acceptable: this technique is of interest only for very small arrays. The overlap technique [12] offers uniform representation and access mechanism as well. It assumes that, for each processor, the location of received elements is close to the local partition and thus extends the allocation of the local array so that it can "house" nonlocal received data. Although it provides uniformity, the overlap technique, as an optimized compilation technique, shows some limitations: it can be efficiently applied to a restricted number of distributions and access patterns and may lead to the replication of the whole array.

2.4 Independence from Program Transformations

A distributed array management independent from the compilation scheme facilitates the coexistence of different compilation techniques. On the contrary, if the choice of a representation is guided by the analysis of a program fragment (typically a loop), it may happen that several layouts (and associated access methods) are used within the scope of one distribution, possibly necessitating data rearrangement or additional computation. The work presented by Chatterjee et al. in [6] illustrates this problem. The access mechanism proposed for local elements is based on a finite state machine (FSM) that has to be computed not only from the distribution parameters but also from the local iteration domain associated with the loop considered. If more than one loop nest is to be analyzed and compiled, although the array layout remains identical, the computation of the table coding the FSM must be performed for each loop nest even if the same distribution applies.

3 Page-driven Array Management

We present here a uniform management for distributed arrays based on paging. This management is designed in order to achieve efficient accesses while avoiding unacceptable memory overhead. It is build only from the decomposition parameters of arrays and can be applied to HPF distributed arrays. Nevertheless, this work is part of the PANDORE II project so we will use the PANDORE syntax to describe it. Ownership computation associated with this management is also explained.

3.1 Principle

The page-driven data management of PANDORE II follows the main addressing scheme of classic paging systems for memory management. In such systems, logical memory space is broken into groups of contiguous elements (pages). Pages have a fixed predetermined size. A hardware support divides a logical address in two parts: a *page number* and a *page offset*. The page number is used as an index into a *page table* that contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address. If the page size is S, a logical address α produces a page number PG and an offset OF by $PG = \alpha \ div \ S$ and $OF = \alpha \ mod \ S$. If the logical address space is larger than the physical address space, virtual memory management features may be added. In this case, accessed pages may not be present all the time in physical memory but temporarily loaded from a swapping device.

As for our concern, we manage variables –i.e. distributed arrays– and not memory; our aim is not to build a shared virtual memory. Moreover, we stay at the software level rather than relying on hardware components. This leads to the following consequences:

- The notion of page fault is here irrelevant because all distant accesses are solved by prior communications. Besides, data are not necessarily communicated page-wise.
- We can define a specific access mechanism for each distributed array, in particular the page size may be different for each array.
- The original address space is multidimensional; therefore we apply a multi to one-dimensional transformation before splitting the resulting space into pages.

3.2 Paging Distributed Arrays

We define a representation and its access mechanism for each distributed array. The multidimensional index space of a given array is linearized by a function $\mathcal{L}$. The linear address space obtained is split into pages of fixed size S. A processor stores only those pages that contain at least one element assigned to it by the distribution or one received element. Depending on the distribution of the array, $\mathcal{L}$ and S, a page may be possessed by one or several processors.

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. The difference between pages containing local elements and pages containing only received elements lies in the way they are allocated and filled, not in the way they are accessed. A tuple (PG, OF) is computed from the initial index vector $(i_0, \ldots, i_{n-1})$ with the linearization function $\mathcal{L}$ and the page size S:

$$PG = \mathcal{L}(i_0, \ldots, i_{n-1}) \; div \; S$$
$$OF = \mathcal{L}(i_0, \ldots, i_{n-1}) \; mod \; S$$

The table of pages TP is stored on each processor. It indicates the base address of each page present in local memory. The offset is added to this base address to obtain the exact location of the element.

The page partitioning is also used for computing owners of elements. A table similar to TP stores, for each page, the numbers of the processors that own this page. This table is present in the local memory of each processor.

3.3 Tuning Parameters

For a given distributed array, the parameters we can tune for paging are the page size S and the linearization function $\mathcal{L}$. The value of these parameters should be defined in order to achieve good performance in terms of time and memory space.

As speed of access is our prior motivation, time consuming operations (division, modulo and multiplication) should be avoided in the computation of the tuple (PG, OF) but also in the application of the function $\mathcal{L}$. This is achieved by introducing powers of two, turning integer division, modulo and multiplication into simple logical operations. Moreover, the array decomposition can be taken into account when fixing the actual value of S and $\mathcal{L}$. Intuitively, we choose S and $\mathcal{L}$ so that pages "follow" the blocks, and are owned by as few processors as possible.

For a more formal definition, let us consider the following PANDORE II array distribution on P processors. It should be noticed that the *block*() distribution of PANDORE II is similar to the CYCLIC(M) directive of HPF, see [1] for a more detailed semantics of PANDORE II distribution specifications.

$$int\ V[h_0]\cdots[h_{n-1}]\ by\ block\ (s_0, \ldots, s_{n-1})\ map \left| \begin{array}{l} regular \\ wrapped \end{array} \right| (d_0, \ldots, d_{n-1})$$

n	number of dimensions of the distributed array	$n > 0$
h_k	size of the array in the k^{th} dimension	$h_k > 0$
s_k	k^{th} parameter of the decomposition function	$1 \leq s_k \leq h_k$
d_k	k^{th} parameter of the mapping function	$(d_k)_0^{n-1} = permut(0, \ldots, n-1)$

We consider the access to an element of V noted $V[i_0]\cdots[i_{n-1}]$. To introduce powers of two, we define the function $\theta_{sup}(n)$ (resp. $\theta_{inf}(n)$) for extending an integer to the smallest (resp. largest) power of two greater (resp. less) than or equal to:

$$\theta_{sup}(n) = 2^\rho \text{ with } 2^\rho \leq n < 2^{\rho+1}$$
$$\theta_{inf}(n) = 2^\rho \text{ with } 2^{\rho-1} < n \leq 2^\rho$$

Prior to the definition of S and $\mathcal{L}$, we choose a particular dimension δ, the dimension in which the block size is the largest. If there are several such dimensions, the one corresponding to a non-distributed array dimension or a block size equal to a power of two is chosen, if possible:

$$\begin{aligned}
\Delta_1 &= \{x / \forall k \in 0, \ldots, n-1 \;\; s_x \geq s_k\} \\
\Delta_2 &= \{x / h_x = s_x \text{ or } s_x = 2^\rho\} \\
&\delta \in \Delta_1 \\
&\delta \in \Delta_2 \text{ if } \Delta_1 \cap \Delta_2 \neq \emptyset
\end{aligned}$$

The page size S is then given by:

$$\begin{aligned}
&\text{if } s_\delta = h_\delta \text{ or } s_\delta = 2^\rho \\
&\quad then \;\; S = \theta_{sup}(s_\delta) \\
&\quad else \;\; S = \theta_{inf}(s_\delta)
\end{aligned}$$

$\mathcal{L}$ is the C linearization function for multidimensional arrays applied to a permutation of the index vector. This permutation puts the index corresponding to dimension δ in last position. Moreover, the array dimensions (coefficients of $\mathcal{L}$) are extended to the next power of two. $\mathcal{L}$ is defined by

$$\mathcal{L}(i_0, \ldots, i_{n-1}) = \sum_{k=0}^{n-1} \left(i'_k \prod_{l=k+1}^{n-1} h'_l \right)$$

where i'_k is the k^{th} access index after permutation, i.e:

$$\begin{aligned}
&i'_{n-1} = i_\delta \\
&\forall k \in 0, \ldots, \delta-1 \;\; i'_k = i_k \\
&\forall k \in \delta, \ldots, n-2 \;\; i'_k = i_{k+1}
\end{aligned}$$

and h'_k is the extended size of the array in the k^{th} dimension, i.e:

$$\begin{aligned}
&h'_{n-1} = \theta\left(\left\lceil \frac{h_\delta}{S} \right\rceil\right) \times S \\
&\text{if } n > 1 \\
&\quad h'_0 = h_0 \text{ if } \delta > 0, \text{ else } h_1 \\
&\quad \forall k \in 1, \ldots, \delta-1 \;\; h'_k = \theta(h_k) \\
&\quad \forall k \in \delta, \ldots, n-2 \;\; h'_k = \theta(h_{k+1})
\end{aligned}$$

For example, the distribution (with one non-distributed dimension)

$$A[200][100][50] \text{ by block}(5, 100, 10)$$

will lead to the following definitions:

$$\begin{aligned} S &= 128 \\ \mathcal{L}(i,j,k) &= (64 \times 128)i + 128k + j \end{aligned}$$

and the distribution (with every dimension distributed)

$$B[500][200] \text{ by block}(100, 10)$$

will give:

$$\begin{aligned} S &= 64 \\ \mathcal{L}(i,j) &= 512j + i \end{aligned}$$

3.4 Optimizing the Computation of (PG, OF)

Unlike with a classic paging mechanism, the explicit computation of the linear address $\mathcal{L}(i_0, \ldots, i_{n-1})$ before its splitting into (PG, OF) is not mandatory because we do not rely on a hardware support that needs a memory address. Besides, this intermediate result may lead to unnecessary operations as in the following example:

$$\begin{aligned} A[100][200] &\text{ by block}(10, 200) \\ S &= 256 \\ \mathcal{L}(i,j) &= 256i + j \end{aligned}$$

The page number and the offset will be obtained by

$$\begin{aligned} PG &= (256i + j) \; div \; 256 \\ OF &= (256i + j) \; mod \; 256 \end{aligned}$$

These expressions could obviously be simplified in $PG = i$ and $OF = j$. To make the simplifications clearly visible, we express directly PG and OF as a function of the index vector.

$$page(i_0, \ldots, i_{n-1}) = (PG, OF)$$

with

$$\begin{aligned} PG &= \sum_{k=0}^{n-2} \left(i'_k \prod_{l=k+1}^{n-1} np'_l \right) + i'_{n-1} \; div \; S \\ OF &= i'_{n-1} \; mod \; S \end{aligned}$$

where np'_k is the number of pages in the k^{th} dimension after permutation:

$$\begin{aligned} np'_{n-1} &= \frac{h'_{n-1}}{S} \\ \forall k \in 0, \ldots, n-2 \;\; np'_k &= h'_k \end{aligned}$$

When dimension δ is not distributed, that is to say when $h_\delta = s_\delta$, index i'_{n-1} (i.e i_δ) is always less than or equal to S, *div* and *mod* can be removed:

$$PG = \sum_{k=0}^{n-2} \left(i'_k \prod_{l=k+1}^{n-1} np'_l \right)$$
$$OF = i'_{n-1}$$

Here is the result of these optimizations for the two examples presented in the previous section:

$A[200][100][50]$ by block$(5, 100, 10)$
$PG = (8192i + 128k + j)\ div\ 128 = 64i + k$
$OF = (8192i + 128k + j)\ mod\ 128 = j$

$B[500][200]$ by block$(100, 10)$
$PG = (512j + i)\ div\ 64 = 8j + (i\ div\ 64)$
$OF = (512j + i)\ mod\ 64 = i\ mod\ 64$

3.5 Page Ownership

Each processor stores the table of owners TO which indicates, for each page, the number of the processor that owns this page. This table can be filled using the function $owner(PG, OF)$ that returns the owner of an element.

$$owner(PG, OF) = map \circ page^{-1}(PG, OF)$$

Function $page^{-1}$, the reverse function of *page*, returns the index vector corresponding to a page number and an offset.

$$page^{-1}(PG, OF) = (i_0, \ldots, i_{n-1})$$

with

$$i_\delta = S \times (PG\ mod\ np'_{n-1}) + OF$$
$$\forall k \in 0, \ldots, \delta-1 \quad i_k = i'_k$$
$$\forall k \in \delta+1, \ldots, n-1 \quad i_k = i'_{k-1}$$
$$\forall k \in 0, \ldots, n-2 \quad i'_k = \left(PG\ mod \left(\prod_{l=k}^{n-1} np'_l \right)\ div \left(\prod_{l=k+1}^{n-1} np'_l \right) \right)$$

Function *map* associates a processor number with an index vector; it depends on the mapping function used in the distribution specification:

- *regular* mapping:
  $map(i_0, ..., i_{n-1}) = \left(\sum_{k=0}^{n-1} \left((i_k\ div\ s_k) \prod_{d_l < d_k} nb_l \right) \right)\ div\ \left\lceil \frac{\prod_{j=0}^{n-1} nb_j}{P} \right\rceil$
- *wrapped* mapping :
  $map(i_0, ..., i_{n-1}) = \left(\sum_{k=0}^{n-1} \left((i_k\ div\ s_k) \prod_{d_l < d_k} nb_l \right) \right)\ mod\ P$

where nb_k is the number of blocks in the k^{th} dimension: $nb_k = \left\lceil \frac{h_k}{s_k} \right\rceil$

The definitions adopted for S and $\mathcal{L}$ allow the number of owners of a page to be less than or equal to two. If the owner of a page is always unique, any valid value of OF can be used for determining the owner of a page. In the case the owner of a page is not unique, we can compute OF_{lm}, the offset from which the owner changes. In this case, the table of owners stores for each page, the two processor numbers plus the limit OF_{lm}.

$$\forall OF \in 0, \ldots, OF_{lm} - 1 \quad owner(PG, OF) = owner(PG, 0)$$
$$\forall OF \in OF_{lm}, \ldots, S - 1 \quad owner(PG, OF) = owner(PG, OF_{lm})$$

with

$$OF_{lm} \;=\; \text{if } \varphi < S \text{ then } \varphi \text{ else } 0$$
$$\varphi \;=\; ((PG \bmod np'_{n-1}) \times (s_\delta - S)) \bmod s_\delta$$

4 Implementation

A full implementation of the data management mechanisms described above has been realized within the PANDORE II environment. Management of tables, pages and accesses to array elements are shared out among the compiler and the runtime library. As all the tables and pages are needed only during the execution of a distributed phase (no inter-phase analysis is performed at this time), the entire memory space allocated is freed at the end of the phase.

4.1 Tables and Pages

All the information needed to fill the tables of owners and the tables of offset-limits is known at compile-time; these tables could therefore be statically defined and declared as constants in the generated code. However, in order not to lengthen the size of the generated code, the compiler produces functions that allocate and fill the tables at runtime, at the beginning of each distributed phase. For each distributed array V, a table of owners `TO_V` is defined. If a page may be possessed by two processors, three tables are needed: the table of the owners of the first part of pages `TO1_V`, the table of the owners of the second part of pages `TO2_V` and the table containing the offset-limits `TL_V`.

The runtime library is also in charge of allocating and filling the tables of pages and pages themselves. The tables of pages and pages that contain local elements are allocated at the beginning of the distributed phase. Since we use a host-node model, local elements are received from the host and sent back to it at the beginning and at the end of the phase if necessary. The management of pages containing received elements depends on the compilation scheme. Basic operations provided by the runtime library are the page allocation and the placement of elements (single elements or segments) into pages.

4.2 Accesses

It is clear that the part of the access process that is done at compile-time must be as large as possible. The compiler translates a reference to an array element $V[I]$, where I is an index vector, into a call to a runtime macro `access(desc_V, PG, OF)` where `PG` and `OF`

are expressions of I. All constant subexpressions have been computed and the optimization described in section 3.4 has been performed. As expected, these expressions contain only additions and constant logical shifts and maskings. The work that remains at runtime is therefore to evaluate the expressions and use the table of pages associated with V to produce the right reference. This can be noted by the C expression `*(TP_V[PG]+OF)`. The runtime library is developed with `cpp` macros that prevent from the computation of the address of the page table corresponding to V, so we can actually generate this code.

4.3 Owner

Determining the owner of an element $V[I]$ is carried out a similar way. The compiler generates a call to a runtime macro `owner(desc_V, PG, OF)`. An access to a table `TO_V[PG]` is sufficient at runtime to find the processor number in the case the owner of a page is unique. If a page may be possessed by two processors, a call to a slightly different macro is produced. The execution of this macro will issue a comparison between `OF` and the offset-limit corresponding to page `PG`:

```
if (OF < TL_V[PG])
    then TO1_V[PG]
    else TO2_V[PG]
```

5 Performances

5.1 Performances of the Distributed Array Management

It is difficult to precisely compare the speed of accesses to distributed array elements with the one of sequential accesses because they both depend on the type of processor and on the optimizations the target compiler can perform. However, it can be seen that the executed code for distributed accesses involves only few basic operations that generate a very small overhead and may even be more efficient thanks to better optimizations.

The following tables give 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 was used in a previous version of PANDORE II; it performs at runtime a modulo and an integer division 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 and on a node of the iPSC/2. Native compilers have been used with no optimization option.

Sparcstation

sequential		*page*		*block*	
best	worst	best	worst	best	worst
0.30	0.42	0.34	0.38	0.48	1.58

iPSC/2 node

sequential		*page*		*block*	
best	worst	best	worst	best	worst
0.94	2.05	2.14	2.26	3.52	9.86

Likewise, the determination of the owner of an array element requires only a few simple operations, so its cost remains very low. As shown in the previous table, it is preferable to exploit the page decomposition, although it seems to be more natural to base the computation of the owner of an element on the computation of the corresponding block number.

The price to pay for speed of access and speed of ownership computation is the need for a larger amount of memory. Overhead is only due to tables because no additional space is required for pages. When a page contains elements that will never be accessed because of the extension of array dimensions, or because the page is shared by two processors, only the potentially accessed part of the page is actually allocated. A translation of the corresponding pointer in the table of pages is performed if the end of the page is allocated.

The memory overhead due to tables is directly linked to the number of pages, which is in general at least of an order of magnitude less than the size of the array. The following table 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	×1.08
double A[100000] by block(1024)	98	25000	588	×1.02
double A[1000][1000] by block(1,1000)	1000	250000	6000	×1.02
double A[1000][2000] by block(50,500)	8000	500000	80000	×1.16
double A[1000][2000] by block(50,512)	4000	500000	24000	×1.05
double A[100][100][100] by block(100,1,50)	10000	250000	60000	×1.24

5.2 Integration in the PANDORE II Environment

The page-driven management for distributed arrays has been integrated in the PANDORE II environment and coexists with the two compilation schemes used by the compiler. The basic compilation scheme, that relies on a runtime resolution technique, can be applied to every input program. The optimized scheme is based on integer programming and linear algebra results; it performs an analysis of parallel loops[10].

We present in figure 1 the results of an experiment with the two compilation schemes. A comparison is made between a block-oriented array management and the page-driven management. The application is a Red-Black Successive Over-Relaxation algorithm run on

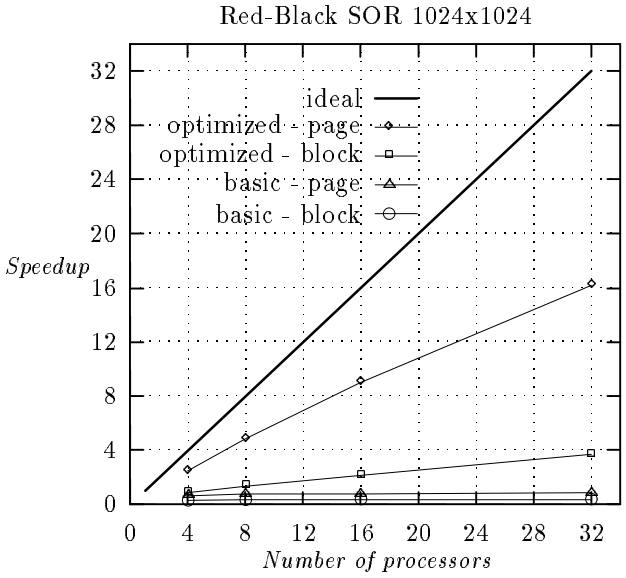

Figure 1: Comparison between block-oriented and page-driven managements

a 1024x1024 matrix of floats. We plotted the speedup against the number of processors for each pair (compilation scheme, array management).

The use of the page-driven management clearly improves performances of codes generated according to both compilation schemes. The joint use of the optimized scheme and the page-driven array management leads to satisfactory performances (more than 50% efficiency for 32 processors) in spite of the unfavorably high ratio of memory operations to computation of the Red-Black.

6 Conclusion

This page-driven management of arrays has proved to be efficient and results presented here may improve on other distributed machines such as the CM5 using more up-to-date processors.

This management is independent from the optimization techniques used in compilers. It avoids using multiple representations of the same array in different parts of a program. The page-driven array manageemnt also seems to be appropriate for irregular computations and could be used together with the inspector/executor technique [7].

We plan to carefully compare our management scheme with shared virtual memory systems and try to find out if HPF compilers can efficiently combine a shared virtual memory with the *owner write rule* or if a specific runtime support is more efficient.

References

[1] F. André, O. Chéron, and J.-L. Pazat. Compiling sequential programs for distributed memory parallel computers with pandore II. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 293–308, Elsevier Science Publishers B.V., 1993.

[2] F. André, J.L. Pazat, and H. Thomas. Pandore a system to manage data distribution. In *International Conference on Supercomputing*, ACM, June 1990.

[3] F. Bodin, L. Kervella, and T. Priol. Fortran-S : A Fortran Interface for Shared Virtual Memory Architectures. In *Proc. of Supercomputing 1993*, novembre 1993.

[4] T. Brandes. Compilating data parallel programs to message passing programs for massively parallel MIMD systems programs. In *Working Conference on Massively Parallel Programming Models, Berlin*, September 1993.

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

[6] S. Chatterjee, J.R. Gilbert, F.J.E. Schreiber, and S.H. Teng. Generating Local Adresses and Communication Sets for Data-Parallel Program. In *The Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, July 1993.

[7] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler methods for irregular problems - data copy reuse and runtime partitioning. In *Third Workshop on Compilers for Parallel Computers*, pages 185–219, Austrian Center for Parallel Computation, July 1992.

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

[9] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.W. Tseng. *An Overview of the Fortran D Programming System.* Technical Report TR91121, Center for Research on Parallel Computation, Rice University, March 1991.

[10] M. Le Fur, J-L. Pazat, and F. André. *Static Domain Analysis for Compiling Commutative Loop Nests.* Technical Report 757, Irisa, September 1993.

[11] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993. Also available as Rice COMP TR93-199.

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

[13] H. P. Zima and B. Chapman. *Compiling for Distributed-Memory Systems.* Technical Report APCP/TR 92-17, Austrian Center for Parallel Computation, University of Vienna, November 1992.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399