Distributed Array Management Scheme for Data-Parallel Compilers

Yves Mahéo, Jean-Louis Pazat
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
Fax: (33) 99 84 71 71
e-mail: *name*@irisa.fr

Abstract

High Performance Fortran and other similar languages have been designed as a means to express portable data parallel programs for distributed memory machines. As data distribution is a key-feature for exploiting the parallelism of applications, a crucial point for data-parallel compilers is their ability to manage efficiently distributed arrays. We present in this paper an innovative method to allocate local blocks and temporaries for received values and to manage the associated access mechanisms. The performance of these access mechanisms is measured and experimental results on the use of this array management within an existing compiler are shown.

Keywords: Distributed memory parallel computers, compilation, data-parallel languages, HPF, runtime, paging, memory management.

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 and loop partitioning, it relies on a shared virtual memory [2]; the second one is based on a user-defined data distributions which are used as a guideline to generate communicating processes [7].

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* [4] to distribute the code. This distribution can be done using the runtime resolution technique in which 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 [14, 12, 11, 10].

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

In this paper we present a new 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 managing distributed arrays. 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 Management of Distributed Arrays

2.1 Key issues

The aim of a distributed array management is to define the way distributed arrays are stored in the local memories and the way elements of these arrays are accessed. These tasks involve both the compiler and the runtime support. It is clear that a trade-off must be achieved 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 (typically achieved by translating an array declaration $A(N)$ into a local allocation $A(N/P)$ where P is the number of processors) associated with global-to-local index conversion involving several costly operations such as *mod* and *div* must be avoided.

In addition to the "classical" concerns that are the memory use and the speed of accesses, we define in the following some properties that we claim to be useful for a distributed array management.

Uniformity

Two kinds of data are to be considered on a given processor: *local* elements – i.e. elements assigned to it by the distribution – and received elements that are temporarily stored in the local memory after communication with another processor. We say that an array management scheme is *uniform* when the representation as well as the access mechanism associated with a distributed array makes no distinction between local and received elements. The main advantage of a uniform scheme is that the compiler does not have to separate purely local computation (involving only local data) from computation that needs distant data. Indeed, even if such a separation is sometimes possible at compile-time, it may induce an important code fragmentation in the case of multiple right-hand-side references; performing the separation at runtime brings about costly ownership computation.

Furthermore, defining a non-uniform scheme that gives too great an importance to local elements, at the expense of received elements is not a good solution because it is likely to be harmful not only for accesses but also for communication of non-local data.

Independence

An *independent* array management is only defined from distribution parameters and in particular does not depend on the code itself nor on compiler analysis on the code. A distributed array management scheme 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 part (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 at runtime. Moreover, this independence facilitates the use of different compilation techniques within a code fragment that contains several loops. One way to achieve this independence (as well as uniformity) is to consider that only global indices appear in the generated code.

Contiguity Preservation

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 (in this case no copying nor packing/unpacking is needed between local representations and communication buffers), vector processors, target code optimization and better cache behavior.

2.2 Related Work

To our knowledge, management of distributed arrays have not been studied independently from compilation techniques in existing HPF compilers.

The first technique of storage for distributed arrays, *the overlap* [13] has been implemented both in the Vienna Fortran Compilation System [14] and in the Fortran D compiler [8, 12]: a single subarray is allocated for local data as well as for received data. This technique provides uniform accesses and preserves memory contiguity but it can be applied to a restricted number of distributions and access patterns and may lead prohibitive allocation when distant data location is not close to the local partition. In this case, the Fortran D compiler may select an alternative storage method (buffers) for received values if it can separate purely local computation and computation needing received values. In Vienna Fortran and in an extended version of Fortran D, a specific management related to loops with irregular array accesses is performed through the use of the inspector/executor technique: the local partition is dynamically extended during the inspector phase so that uniform indirect accesses can be used during the executor phase [6, 3, 14].

Other array management schemes, closely related to compilation techniques, have been proposed. Their common characteristic is that they try to minimize memory overhead. Among them, in the compilation scheme defined by Ancourt *et al* [9], 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 evaluating affine functions and possibly integer division. Chatterjee *et al.* [5] 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. Therefore, independence from the compilation scheme is not achieved by these systems.

3 A Page-driven Array Management

We present here a new management scheme for distributed arrays based on software paging. This management is designed in order to achieve efficient accesses while avoiding unacceptable memory overhead. It also aims at satisfying the properties of uniformity, independence and contiguity preservation aforementioned. In the following, we will consider only direct HPF distributions, the mechanisms described can be easily extended to aligned distributions (as a first step, by applying paging to templates).

3.1 Principle

The page-driven data management we propose 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 secondary storage system by 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 only consider compiler-generated code, hence we stay at the software level rather than relying on hardware components. Contrarily to system-level paging,

- 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.
- The original address space is multidimensional; therefore we apply a multi to one-dimensional transformation before splitting the resulting space into pages.

This allows us to define a specific access mechanism for each distributed array, in particular the page size may be different for each array.

3.2 Paging Distributed Arrays

We define a representation and its access mechanism for each distributed array by a couple ($\mathcal{L}$, S). 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.

One of the main advantages of this method 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 allocated locally, no matter if the element it needs to access is truly local or has been received from another processor. The difference between pages containing local elements and pages containing 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:

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

A table of pages 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 similar table, present in the local memory of each processor, stores for each page the numbers of the physical processors that own this page.

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 are 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) are avoided in the computation of the tuple (PG,OF) but also in the application of 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 HPF array distribution:

```
      REAL V(0 : h_0 − 1, ··· , 0 : h_{n−1} − 1)
!HPF$ DISTRIBUTE V(CYCLIC(s_0),...,CYCLIC(s_{n−1}))
```

and the access to an element of `V` noted `V`$(i_0, \cdots, i_{n-1})$. This distribution decomposes the array `V` into rectangular blocks of size $s_0 \times, \cdots, \times s_{n-1}$. We consider the most general distribution directive `CYCLIC(`k`)`. Note that `BLOCK(`k`)` and `CYCLIC(`k`)` distributions are strictly equivalent as far as decomposition is concerned.

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. The page size S is then given by:

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

where $\theta_{sup}(x)$ (resp. $\theta_{inf}(x)$) extends an integer to the smallest (resp. largest) power of two greater (resp. less) than or equal to x.

$\mathcal{L}$ 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:

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

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

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

Here are two examples of definition of S and $\mathcal{L}$; first when there is one non-distributed dimension and second when all the dimensions are distributed:

```
      REAL A(0:199, 0:99, 0:50)
!HPF$ DISTRIBUTE A(CYCLIC(5), *, CYCLIC(10))
```
$$
\Rightarrow \left\{ \begin{array}{l} S \; = \; 128 \\ \mathcal{L}(i,j,k) \; = \; (64 \times 128)i + 128k + j \end{array} \right.
$$

```
      REAL B(0:499, 0:199)
!HPF$ DISTRIBUTE B(CYCLIC(100), CYCLIC(10))
```
$$
\Rightarrow \left\{ \begin{array}{l} S \; = \; 64 \\ \mathcal{L}(i,j) \; = \; 512j + i \end{array} \right.
$$

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:

```
      REAL A(0:99, 0:199)
!HPF$ DISTRIBUTE A(CYCLIC(10),*)
```
$\Rightarrow \begin{cases} S = 256 \\ \mathcal{L}(i,j) = 256i + j \end{cases}$

The page number and the offset will be obtained by

$$PG = (256i + j)\ div\ 256$$
$$OF = (256i + j)\ mod\ 256$$

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

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

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

$$np'_{n-1} = \frac{h'_{n-1}}{S}$$
$$\forall k \in 0, \ldots, n-2 \quad np'_k = h'_k$$

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:

```
      REAL A(0:199, 0:99, 0:49)
!HPF$ DISTRIBUTE A(CYCLIC(5), *, CYCLIC(10))
```
$\Rightarrow \begin{cases} PG = (8192i + 128k + j)\ div\ 128 \\ \quad\ = 64i + k \\ OF = (8192i + 128k + j)\ mod\ 128 \\ \quad\ = j \end{cases}$

```
      REAL B(0:499, 0:199)
!HPF$ DISTRIBUTE B(CYCLIC(100), CYCLIC(10))
```
$\Rightarrow \begin{cases} PG = (512j + i)\ div\ 64 \\ \quad\ = 8j + (i\ div\ 64) \\ OF = (512j + i)\ mod\ 64 \\ \quad\ = i\ mod\ 64 \end{cases}$

3.5 Page Ownership

Each processor stores a table of owners that indicates, for each page, the number of the physical 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

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

Function $map(i_0, \ldots, i_{n-1})$ associates a physical processor number with an index vector. It can be easily computed for each mapping that can be expressed in HPF. We do not give here the general formula as it cannot be induced from the HPF norm and depends on implementation choices. As an example if the abstract processor array is of size $(p_0, \cdots, p_{n-1})$ and the number of physical processors is P, the mapping function may be the following:

$$map(i_0, ..., i_{n-1}) = \left(\sum_{k=0}^{n-1} \left((i_k \; div \; s_k) \prod_{l=k+1}^{n-1} p_l \right) \right) \bmod P$$

In the case data elements are replicated (by application of alignment directives), this function could return a set of processors.

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}:

$$OF_{lm} = \text{if } \varphi < S \text{ then } \varphi \text{ else } 0$$

with

$$\varphi = ((PG \bmod np'_{n-1}) \times (s_\delta - S)) \bmod s_\delta$$

4 Implementation

A full implementation of the data management mechanisms described above has been realized within the PANDORE environment [1]. In the PANDORE language, the scope of array distributions is confined within procedures referred to as *distributed phases*.

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. 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. 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 (`TP_V`) to produce the right reference. This can be noted by the C expression `*(TP_V[PG]+OF)`. The runtime library contains `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]
```

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

Table 1: Speed of page-driven access

5 Performances

5.1 Performances of the Distributed Array Management

It is quite obvious 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.

In the experiment whose results are reported in table 1, we measured the time taken by several kinds of read accesses:

- t_s : a reference to an element as it may appear in a sequential program;
- t_p : a call to the macro that uses the paged access mechanism;
- t_b : a call to a macro that uses a block-oriented access mechanism[1].

The array is a two-dimensional array of floats; reported times are in μs. Best and worst cases have been considered, depending on whether the sizes of the array were powers of two or not. Experiments have been carried out on a SparcStation 2, on a node of the iPSC/2 and on a node of the Intel Paragon XP/S.

Likewise, the determination of the owner of an array element requires only a few simple operations, so its cost remains very low. It is also 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. 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

[1] This mechanism was used in a previous version of PANDORE; it performs at runtime a modulo and an integer division to find the block number and the offset in the block.

Array Distribution	*Number of Pages*	*Minimal Partition*	*Local Space for Tables*	*Local Overhead*
REAL(KIND=8) A(0:99999) !HPF$ DISTRIBUTE A(CYCLIC(1000))	196	25000	1960	×1.08
REAL(KIND=8) A(0:99999) !HPF$ DISTRIBUTE A(CYCLIC(1024))	98	25000	588	×1.02
REAL(KIND=8) A(0:999, 0:999) !HPF$ DISTRIBUTE A(CYCLIC(1), ∗)	1000	250000	6000	×1.02
REAL(KIND=8) A(0:999, 0:1999) !HPF$ DISTRIBUTE A(CYCLIC(50), CYCLIC(500))	8000	500000	80000	×1.16
REAL(KIND=8) A(0:999, 0:1999) !HPF$ DISTRIBUTE A(CYCLIC(50), CYCLIC(512))	4000	500000	24000	×1.05
REAL(KIND=8) A(0:99, 0:99, 0:99) !HPF$ DISTRIBUTE A(∗, CYCLIC(1), CYCLIC(50))	10000	250000	60000	×1.24

Table 2: Memory overhead for some common distributions

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.

5.2 Integration in the Pandore Environment

The page-driven management for distributed arrays has been integrated in the PANDORE environment and is used with the two compilation schemes of 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 table 3 the results of the execution of a Red-Black Successive Over-Relaxation algorithm run on a 1024x1024 matrix of floats. Times have been measured on the Intel iPSC/2 for the two compilation schemes. A comparison is made between a block-oriented array management and the page-driven management. The table shows the speedup obtained on P processors for each pair (compilation scheme, array management).

P	*Basic scheme*		*Optimized Scheme*	
	Block	*Page*	*Block*	*Page*
4	0.29	0.68	0.86	3.84
8	0.32	0.77	1.39	7.18
16	0.36	0.82	2.14	12.78
32	0.37	0.85	3.70	23.72

Table 3: Comparison between block-oriented and page-driven managements

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 (efficiency of around 75% for 32 processors) in spite of the unfavorably high ratio of memory operations to computation of the Red-Black algorithm.

6 Conclusion

Management of distributed arrays is a crucial point for obtaining good performances when using data parallel compilers. For this purpose, we have proposed a new scheme based on parameterized software paging that proved efficient in an existing compiler. This management handles local and received data in an uniform way and it is independent from the optimization techniques used in compilers. Moreover, it avoids using multiple representations of the same array in different parts of a program and maintains some regularity in local layouts. The page-driven array management also seems to be appropriate for irregular computations and could be used together with the inspector/executor technique.

We are currently comparing our management scheme with shared virtual memory systems. One of our objectives is to find out which features should be added to existing shared virtual memory systems so they can be efficiently targeted by data-parallel compilers.

References

[1] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *HPCN Europe '95*, LNCS, Springer Verlag, Milan, Italy, May 1995.

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

[3] P. Brezany, O. Chéron, K. Sanjari, and E. van Kronijnenburg. Processing Irregular Codes Containing Arrays with Multidimensional Distributions by the PREPARE HPF Compiler. In *HPCN Europe '95*, LNCS, Springer Verlag, Milan, Italy, May 1995.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] F. Irigoin, C. Ancourt, F. Coelho, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In H. J. Sips, editor, *Fourth International Workshop on Compilers for Parallel Computers*, pages 117–132, TU Delft, The Netherlands, December 1993.

[10] M. Le Fur, J.-L. Pazat, and F. André. Commutative Loop Nests Distribution. In H. J. Sips, editor, *Fourth International Workshop on Compilers for Parallel Computers*, pages 345–350, TU Delft, The Netherlands, December 1993.

[11] Q. Ning, V. van Dongen, and G.R. Gao. Automatic Data and Computation Decomposition for Distributed Memory Machines. In *Proc. of the 28th Hawaii International Conference on System Sciences*, Wailea, Hawaii, January 1995.

[12] 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.

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

[14] 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.