State of the Art in Compiling HPF*
TR EMP CRI A-286

Fabien COELHO (CRI, École des mines de Paris)
Cécile GERMAIN (LRI, Université Paris-Sud)
Jean-Louis PAZAT (IRISA, Rennes) †

June 1996

Abstract

Proposing to the user a nice programming model based on the data-parallel paradigm is one thing. Running the resulting applications very fast is the next issue for a language aiming at *high performance* on massively parallel machines. This paper discusses the issues involved in HPF compilation and presents optimization techniques, targeting the message-passing SPMD programming model of distributed memory MIMD architectures.

1 Introduction

Parallelism is becoming a ubiquitous technique: micro-parallelism inside processors, moderate parallelism inside workstations, and soon parallel personal computers. Paradoxically, the decline and fall of massive parallelism is often announced. Massively parallel architectures will be killed by shared memory machines, and by fast networks of workstations, because they are too costly and not reactive enough to technology evolution. On the other hand, parallel machines often remain the top performance providers, and at a better cost (by performance unit) than workstations.

The HPF effort may be seen as an attempt to make massively parallel machines usable for their natural applications, which are in the field of scientific computing and number crunching, thus making them attractive for end-users. However, the application field puts heavy pressure on the compiler effort: the users require efficiency and usability, at the same level as what is found on a workstation. But compilers and run-time support for workstations have a long

*PRS Spring School, March 1996, Les Menuires, France, Springer-Verlag LNCS (to appear)
†`Coelho@cri.ensmp.fr, Cecile.Germain@lri.fr, Jean-Louis.Pazat@irisa.fr`

(in time) and large (in involved people) history. HPF was designed in 1993, at a time when the compiler effort was not finished for its predecessor Fortran 90.

At the present time, HPF's usability *and* efficiency goals cannot be fulfilled both for all applications. Regular applications, such as dense linear algebra, enjoy the full benefit of HPF without excessive constraint on the programming style. However, HPF is so rich that, even in this case, the programmer must be aware of the general principles of parallel compiler technology in order to avoid some pitfalls. Moreover, many theoretical results are known, but few have been implemented in commercial compilers. Irregular or dynamic applications are in a poorer state. These applications heavily rely on run-time support. In general, such support has been developed in the message-passing framework. Plugging these tools into HPF compilers is an active research area. This paper will show that an application must present sufficient static knowledge for compiler optimization. Applications that are fully dynamic and irregular, as N-body or tree-code methods, cannot be efficiently ported to HPF just as is. In fact, it is not obvious that they can be efficiently ported to HPF at all, if the target architecture is a classical massively parallel message-passing machine, because such applications are highly adaptive, thus change at each step the distribution of the data and the communication patterns. Between these two extremes, a large number of applications need varying recoding efforts, but may be expected to run efficiently on future HPF compilers.

Parallel compiler technology is rapidly evolving. Hence, this paper focuses on some basic problems related to the really new features of HPF: general parallel assignment and data placement functions. The only target addressed here is the message-passing programming model. A number of issues for compiling HPF are first outlined (Section 2). Then an overview of HPF optimized compilation techniques and their formalization in an affine framework is presented. The existence of such a framework is the key to understanding the underlying regularity displayed by HPF mappings. The enumeration techniques for regular (Section 4) and irregular (Section 5) accesses are described. Finally the allocation techniques (Section 6) suggested so far are presented, illustrated and discussed.

2 HPF Compilation Issues

Let us introduce the three main issues depicted in Figure 1. On top is the application the user has in mind. This application is embedded in the data parallel paradigm and implemented using the language. The user suggests through directives the mapping of data onto an abstract distributed memory parallel machine, and also gives hints about the parallelism available in the application. Detecting the parallelism and choosing a good mapping automatically is an active research area and is beyond the scope of this paper. From the HPF code and directives, the compiler is expected to generate some efficient code for the target architecture. All layers of the process contribute to problems for the compiler.

2.1 Application

The application is hopefully parallel to a large extent, and presents some kind of locality and repeated reference patterns, so that the communication induced by the distribution of data is small compared to the computations. For some applications the data parallelism may be large, but the problems are irregular and dynamic and the translation into the simple data structures available in Fortran, and the regular HPF mappings cannot express the actual locality. Such irregular problems must rely on advanced runtime librairies for analyzing and reusing communication patterns. More precisely, an application is *regular* if the references to distributed arrays are affine functions of the loop bounds. An application is *irregular* if the references to distributed arrays are data-dependent. Finally, an application is dynamic if its data structures evolve during the computation: in general this happens for irregular applications where the data used in the references to distributed arrays change.

2.2 Language

[26] on which HPF is based, includes many features and intrinsics. Several years were necessary for the first compilers to be available. Also *old* features inherited from Fortran 77 [4], such as commons, storage and sequence association, and assumed size arguments, are closely linked to the idea of a sequential machine and contiguous memory and cannot be implemented simply on a distributed memory machine because the compiler does not know about the real structure intended by the user. On this large base were added HPF extensions [16]. Correctness must be addressed, that is every Fortran feature must coexist with every HPF feature. Tricky details must be systematically handled that are not related to performance but require energy and manpower.

The second problem is the variety of HPF mappings allowed, through alignments and distributions. Alignment strides and general cyclic distributions may lead to unexpectedly strange and apparently irregular mappings, as shown in Figure 2 where shaded elements show the array elements mapped onto the corresponding processor. One processor gets all the elements despite the cyclic distribution in one example; in the other, the even processors get twice as many array elements as the odd ones. Moreover, these possibly complex mappings, sometimes involving replication, may not be known at compile time, because of ambiguity allowed by remappings (Figure 3) or because they may depend on runtime values (for instance, the number of available processors), and also because of the `inherit` directive (Figure 4) at a procedure interface.

Application	Regularity Locality Parallelism
Language	HPF extensions Fortran 90 Fortran 77
Target	Programming model Runtime support Architecture

Figure 1: 3 layers view

```
      real A(n)
chpf$ processors P(4)
chpf$ template T(2*n)
chpf$ align A(i) with T(2*i)
chpf$ distribute T(cyclic(3)) onto P
```

```
      real A(n)
chpf$ processors P(4)
chpf$ template T(4*n)
chpf$ align A(i) with T(4*i)
chpf$ distribute T(cyclic) onto P
```

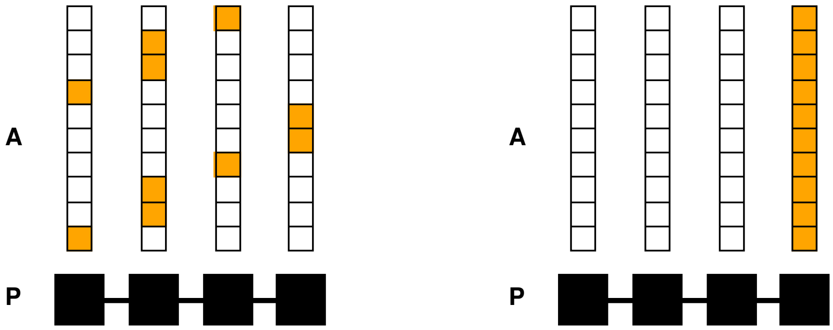


Figure 2: Examples of strange mappings allowed by HPF

```
chpf$ distribute A(cyclic)
      if (A(1).lt.0.) then
chpf$   redistribute A(block)
      endif
! now the compiler cannot know whether
! A is block- or cyclic-distributed.
      B(:) = A(:)
```

Figure 3: Ambiguity of re-mappings

```
      subroutine foo(A)
      real A(10)
chpf$ inherit A
! the compiler cannot know about
!   A mapping.
      A(:) = ...
      ...
```

Figure 4: `inherit` Un-knowledge

```
  SEQUENTIAL           MESSAGE-PASSING                  GET PUT SYNCHRO
! x on P1         ! P1          ! P2             ! P1                ! P2
! y on P2         t = rcv P2    snd y to P1      t = get y on P2
x = f(y)          x = f(t)                       x = f(t)
y = g(x)          snd x to P2   t = rcv P1       put x as s on P2
                                y = g(t)         synchro             synchro
                                                                     y = g(s)
```

Figure 5: Message-Passing/Get Put Synchronize

2.3 Target

Different target programming models and architectures require different techniques and optimizations. First, SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data) architectures may be targetted. For MIMD architectures, a (Single Program Multiple Data) program parametrized by the processor identifier must be generated. The rationale for generating few programs (typically one or two) to be run identically in parallel on the nodes of the target machine is obvious: it is the only scalable (*i.e.* extensible to a large number of processors) technique. The underlying communication model may be message passing *à la MPI* or a get, put and synchronize model (Figure 5), requiring a different style of optimization. The communication to computation speed ratio as well as the communication latency and bandwidth often differ, changing the optimal choice for compilation. Some machines have special hardware for particular communication patterns (broadcasts, synchronizations, reductions, shifts) which is significantly more efficient [36] than simple point to point communications, so that an efficient compiler *must* use them. The diversity of target architectures results in a necessary diversity or adaptability of the compilation techniques.

2.4 Outline

This paper focuses on issues related to the efficient management of HPF's very general regular mappings, allowing alignment strides and general cyclic distributions for distributed memory MIMD architectures. The compiler must switch from the global name space and implicit communications of HPF to the local name space and explicit communications of the target SPMD message-passing programming model. The efficiency of the allocation, addressing, communication and temporary managements are the key points for achieving acceptable performance.

3 Compilation overview

Let us focus on the generic example in Figure 6 where `A` (lhs) is assigned and `B` (rhs) is used within a parallel loop. Iteration set $\mathcal{I}$ and access functions g_1 and g_2 may be arbitrary. The correctness of the parallelism requires that f

```
independent forall (i ∈ I) A(f(i)) = B(g1(i)) + B(g2(i))
```

$$\texttt{independent forall}\ (i \in \mathcal{I})\ \texttt{A(}f(i)\texttt{) = B(}g_1(i)\texttt{) + B(}g_2(i)\texttt{)}$$

Figure 6: Generic parallel assignment

Processor 0	Processors 1-2	Processor 3
	send y(0)	send y(0)
do j = 0:2	do j = 0:2	do j = 0:2
x(j) = y(j+1)	x(j) = y(j+1)	x(j) = y(j+1)
receive(temp)	receive(temp)	
x(3) = temp	x(3) = temp	

(a)

Processor p
if (p ≠ 0) send y(0)
do j = 0:2
x(j) = y(j+1)
receive(temp)
if (p ≠ 3) x(3) = temp

(b)

Figure 7: Hand-made and SPMD code with OCR

is injective on $\mathcal{I}$. From such a loop the compiler must (1) assign iterations to processors, (2) determine the resulting communications and (3) generate code for the SPMD message passing model. If these tasks cannot be performed by the compiler, they are delagated to the runtime system, with the inefficient but necessary *runtime resolution* technique. Each point is discussed in the following.

3.1 Compiling for the SMPD model

When considering the SPMD execution model as the target of the compilation process, the most important fact is that the generated code must be *generic*: the code is identical on all processors, necessarily parametrized by the processor identifier. When considering distributed memory architectures as the target of this SPMD code, the most important characteristic is the : access to remote data is much slower than access to local data. Finally, when considering a message passing model for communication, the memory hierachy has qualitative effects: access to remote data requires knowing which processor owns the data. From all these features, the compiler makes fundamental choices:

- mapping of computations, that is assigning iterations ($\mathcal{I}$) to processor(s);
- allocating a local representation for the distributed data on each processor;
- code generation scheme, involving the actual translation from global to local adresses and the scheduling of communications with respect to computations.

3.2 Owner Computes Rule

For assigning computations to processors, compilers generally rely on the (*OCR*). The rule states that the *computation* for a given array assignment must be performed by the *owner* of the assigned reference. This simplifies the generated code: once a value is computed on a processor, it can be stored directly without further communication. The *OCR* is not necessarily the best choice and possible alternative solutions have been investigated [7]. Making this choice visible to some extent to the user is proposed in some languages, with the ON clause. Given the assignment, the compilation problems are basically identical with all schemes, hence we assume the *OCR* in the following.

Figure 7 examplifies the application of the *OCR* to the compilation process. Arrays X and Y are supposed to be distributed `block(4)` onto four processors. In both cases, x and y are the local arrays implementing the global arrays X and Y, and are accessed by local addresses. The code in (a) is tailored to the specific processors, not SPMD. The generic code in (b) involves some guards.

3.3 Run-time resolution

The simplest compilation scheme [9] uses of accesses to distributed data (*RTR*). Each processor executes the whole control flow of the program, but each reference to distributed data is guarded by a locality check. Communication is then generated between the owner and any processors that may need the data (see Figure 8). This technique can be implemented simply [12] as a rewriting of the syntax tree of the original program.

Run-time resolution is highly inefficient, and the inefficiency increases with the degree of parallelism. First, each process scans the entire iteration space and executes the whole control flow of the program. Worse, for each index, a guard is evaluated; the guard is pure overhead compared to the sequential execution. Next, as the scheme is based on global indices, a large number of address translations from global to local indices must be performed at run-time. Finally, the communication is not vectorized but performed in an elemental level. Checking ownership is simple when addresses are explicit functions of loop indices, for instance affine functions; from i, $f(i)$ is computed, and the ownership is derived from the formulas in Figure 12. When adresses are themselves distributed array references, the *RTR* scheme must be applied twice.

Clearly, optimizations are necessary. However, possible optimizations will depend heavily on the nature of the application. If the references are regular, aggressive compilation analysis is possible. Basically, a large fraction of the guards and address translations will shift from execution time to compile time. For irregular applications, the run-time resolution overhead can be amortized either over repeated executions of a parallel loop, for instance if the parallel loop is enclosed in a sequential one, or across code parts that require the same address translations. It must be stressed that, if an application is essentially dynamic and irregular, it cannot benefit from more than a limited number of optimizations, with respect to run-time resolution. Therefore choices have to be devised at the algorithmic level, or even at the application level. For instance, in the field of parallel finite elements problems, if one asks for programming in

HPF, domain decomposition methods may be easier and more effective than global sparse computations.

```
// source code: A[i] = f(B[j])

Oa = owner(A[i]);
Ob = owner(B[j]);
if (Oa==me) {
  tmp = (Ob==me)? B[j]: recv(Ob);
  A[i] = f(tmp); // computation!
} else {
  if (Ob==me) send(B[j],Oa);
}
```

notations:

- `me`: processor identifier
- `tmp`: temporary local data
- `owner(x)`: the owner of `x`
- `send(x,p)`: non block. send `x` to `p`
- `recv(p)`: blocking receive from `p`

Figure 8: Runtime resolution on a single statement

3.4 Index and communication sets

The compiler decision for the *OCR* assigns loop iterations to processors. This can be formalized into a parametric set of iterations, and the resulting communications into sets of array indices that must be communicated between processors. For clarity, the generic example in Figure 6 where `A` (lhs) is assigned and `B` (rhs) is referenced within a parallel loop is considered. An example is presented in Figure 13. For each processor and array the mapping directives define a set of owned array references:

$$\mathcal{O}_{\mathtt{A}}(p) = \{a | \mathtt{A}(a) \text{ is mapped on } p\}$$

Each processor is assigned a set of iterations to compute with the *OCR*:

$$\mathcal{C}(p) = \{i \in \mathcal{I} | f(i) \in \mathcal{O}_{\mathtt{A}}(p)\}$$

In turn, this set defines the set of references needed by a processor to perform the computation, that is the set of viewed elements. In our example:

$$\mathcal{V}_{\mathtt{B}}(p) = \{a | \exists i \in \mathcal{C}(p), a = g_1(i) \text{ or } a = g_2(i)\}$$

From these set one can define the sets of array references that are to be communicated from processor p to processor p':

$$\mathcal{S}_{\mathtt{B}}(p, p') = \mathcal{O}_{\mathtt{B}}(p) \cap \mathcal{V}_{\mathtt{B}}(p')$$

3.5 Code Generation

Several variations are possible in defining the previous high level sets, each directed towards a particular scheme. In any case, the SPMD code will have to

send $\mathcal{S}(p,*)$
receive $\mathcal{S}(*,p)$
for $i \in \mathcal{C}(p)$ ...

Figure 9: Simple SPMD code

send $\mathcal{S}(p,*)$
for $i \in \mathcal{C}_{\text{local}}(p)$ ...
receive $\mathcal{S}(*,p)$
for $i \in \mathcal{C}_{\text{remote}}(p)$ ...

Figure 10: Overlap SPMD code

scan these sets, that is enumerate all their elements. The simple code presented in Figure 9 performs first the communication and then the computations ($*$ stands for *all possible values*). The code presented in Figure 10 from [31] overlaps local computations and communication by splitting $\mathcal{C}$. Computations involving *remote* references are delayed. The purpose of this split scheme is *latency hiding*, by performing the *local* computations while the messages are being transmitted on the network. However, it is shown in Section 6 that the split scheme creates other penalties that may exceed its benefits.

In both schemes, the messages may be built one after the other for each target processor, or all at the same time; the latter may simplify the enumeration part at the price of having to wait for all the messages to be ready before issuing the actual communication. This distinction also holds for the receiving messages. Thus the very same sets and high level SPMD code can lead to very different communication behavior.

Parallel loops more general than the example of Figure 6, in particular forall loops including many statements, may be split into a sequence of parallel one-statement loops, or compiled as a whole. In the latter case, the iteration set $\mathcal{C}$ is fundamental. The union of the statement sets of each statement in the loop forms the Reduced Iteration Set (RIS) of the loop [39]. If the $\mathcal{C}$ of a particular statement in the loop is a proper subset of the RIS, then the corresponding statement must be guarded.

In [35] a general calculus for building these sets is presented. However the main issue is the practical management of these parametric sets at compile- and run-time. What data structure? What operators? Cost and approximations? Code generation issues? The choice of data structures ranges from array sections to polyhedra for regular references, and includes lists for irregular applications. It should be noted that the more general the data structure is, the more expensive the operators are, but also the less restrictions are needed on $\mathcal{I}$, f, g_1 and g_2. A typical restriction is that for general polyhedron-based techniques [2, 33] the iteration space and references are described by affine functions.

Desirable features for such a data structure are:

- minimality with respect to the described set, to avoid guards;
- minimality with respect to the number of operations involved; in particular, if an array reference of the SPMD code is expressed in the native global framework, it requires an expensive runtime translation to local

addresses.

- closure under common operators (union, intersection, difference ...)
- easy to scan in the SPMD programming model; loops are the best choice, lookup tables must be kept to a moderate size;

It is not at all obvious that a perfect data structure exists. Thus all compilation techniques must make choices between these properties, and assume the implied approximation and other costs.

3.6 Optimizations

Overhead in SPMD code with respect to sequential code has two sources: extra computation and control, and communication. The first issue is tackled in the next section; this section sketches . Clearly, communication is pure overhead and must be treated with special care, because of its overwhelming cost. Optimization occurs at two levels: local optimization deals with the communication created by a given parallel loop; global optimization considers the analysis and scheduling of communications across the whole code, at the intra- and inter-procedural level.

At the local level, because the start-up cost is much higher than the per-element cost of sending a message, a major goal for a compiler is to combine messages from a given source to a given destination. For parallel loops, this results in moving communication outside the loop. This leads to three optimizations [24] . *Message vectorization* packs the references associated with the same right-hand-side expression. *Message coalescing* packs the references associated with different right-hand-side expressions. *Message aggregation* allows reuse of previously sent data. Vectorization and coalescing reduce the number of messages, while aggregation diminishes the overall volume of communication. It has been suggested [19] to extend the concept of vectorization to collective communication.

A comparative study of the respective impact of each optimization has been made [41]. Vectorization was found to be the most important, and the most consistent across the benchmarked codes. Also coalescing is very useful to typical communications arising from references to a same array in stencil computations. However, some applications may also benefit a lot from aggregation. For instance, some molecular dynamics code use lists of interacting particles. As interaction is guided by physical distance, if particles A and B interact with particle C, they will probably interact with each other. If particles A and C are mapped on the same processor and B is remote, the data for B will be sent to interact with C, and does not need to be sent once more for A.

The placement of communication within the program answers the following question: where should a communication take place in order to minimize the

```
      real A(0:42)
chpf$ template T(0:127)
chpf$ processors P(0:3)
chpf$ align A(i) with T(3*i)
chpf$ distribute T(cyclic(4)) onto P
      A(0:42:3) = ...
```

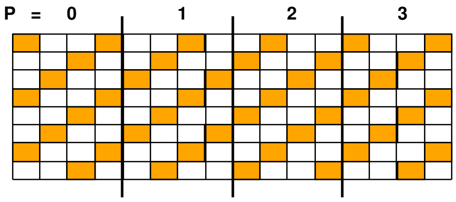


Figure 11: Mapping example from [11]

overall communication, remove useless communication, and minimize latency. Consider the code fragment:

```
A(:,5) = B(:)
forall (j=1:10) A(:, j) = B(:)
```

Without analysis, two groups of communication is necessary: the first statement copies `B` to the owner of column 5 of `A`, and the loop copies `B` on each column of `A`. If the second communication is moved above the first assignment, the first communication may be removed. A data-flow approach is well-suited to this problem [1, 42, 10, 20]. Through advanced program analyses, the source of a value in a program is identified (last write tree, array data flow graph) and the communication is issued accordingly.

3.7 Affine framework

Let us introduce the general mapping example in Figure 11. Array `A` is mapped onto a processor arrangement `P` through a stride 3 alignment and a general cyclic-4 distribution. The picture shows the resulting mapping of the array (shaded elements) and the template onto the 4 processors. The 128 template elements cycle over the processors: each row is a cycle of 16 template elements. Contiguous blocks of 4 elements are attached to each processor. For instance `T(18)` is on processor $\psi = 0$, at cycle $\gamma = 1$ and offset $\delta = 2$.

From this picture the mapping geometry of array elements seems regular. This regularity comes from the affine constraints that can be derived:

`A(0:42)`	$0 \leq \alpha \leq 42$
`T(0:127)`	$0 \leq \theta \leq 127$
`P(0:3)`	$0 \leq \psi \leq 3$
`align A(i) with T(3*i)`	$\theta = 3\alpha$
`cyclic(4)`	$0 \leq \delta \leq 3$
`distribute T(cyclic(4)) onto P`	$\theta = 16\gamma + 4\psi + \delta$

Integer solutions to this set of equalities and inequalities provide an exact description of the HPF mapping, that is the array element (α), the processor onto which it is distributed (ψ), the cycle it belongs to (γ) and so on. The first inequalities just reflect the Cartesian declaration constraints. Then the alignment constraint links the array element α to its corresponding template cell θ. The block width is expressed on the block offset δ. Finally the distribution equation decomposes the template space θ into the corresponding processor ψ, cycle γ and offset δ.

Such an affine formalization is presented in [2]. All mapping issues, including alignment strides and cyclic distributions as in this example, but also partial replication and multidimensional distributions can be included. Moreover, the elements accessed by an array section (independent affine functions of one index) and also by some loop nests, as well as the allocation and addressing issues on a given processor, can be described within the same framework. For the array assignment in Figure 11, after introducing index variable i, we have:

`0:42:3`	$0 \leq i \leq 14$
`A(0:42:3)`	$\alpha = 3i$

global to distributed:	$\gamma = \lfloor \theta/pk \rfloor, \psi = \lfloor \theta \bmod pk/k \rfloor, \delta = \theta \bmod k$
distributed to global:	$\theta = pk\gamma + k\psi + \delta$

Figure 12: HPF distribution functions

The very simple example of expressing the distribution function shows the benefits of an affine framework. Figure 12 gives the conversion formulas for an array indexed by θ distributed cyclic(k) on a p processor set (all arrays are considered indexed from 0). Often, the compilation process will have to translate the θ framework into the (γ, ψ, δ) framework. The simplest example is applying the *OCR*, but computing local addresses is also closely related to this translation. For this purpose, the formulas in the second column are not convenient, because symbolic computations using integer division are restricted. An is a practical alternative: the equation in the first column describes the same $(\theta, \gamma, \psi, \delta)$ set as the equations of the second column, if we add that δ and ψ are bounded ($0 \leq \delta \leq k-1$ and $0 \leq \psi \leq p-1$). Instead of constraining the way the compiler must translate global addresses to local addresses, the constraints are expressed in a non directive way. The compiler can take advantage of this to compute the local addresses so as to respect these constraints, but not necessarily to start its computation from global addresses.

3.8 The alignment/distribution problem

When an array is its own template, it has a regular HPF distribution (**`cyclic(k)`** or **`block(k)`**). Allocating the corresponding memory space is easy if the processor extents are known at compile time: roughly, the local array size is the block size times the number of cycles. Simple and efficient local addressing schemes can be derived.

Problems come from alignment. In a pseudo-mathematical formulation, one can state that HPF distributions are not closed under alignment: combining an HPF distribution and an HPF alignment does not always result in an HPF distribution. Figure 11 and Figure 2 show such examples: the distributions may be considered as an irregular (but static) mapping, because the block size varies from processor to processor, and worse, inside each processor.

What new issue does these complex distributions raise? The code generation issues are solved by enumerating complex sets at runtime to perform computations and communications. However, even if such techniques can be reused, the problem is actually new: here the compiler must *allocate* memory space for these irregular blocks. The allocation scheme may be optimal in memory space: all the elements of an array placed on a given processor are allocated contiguously, possibly at the expense of efficiency in physical address computation. At the other extreme, one can consider allocating the whole template, then wasting memory space, but resuming to a regular case. Typical allocation schemes are detailed in Section 6.

4 Regular References

The sets describing computations and communications presented in Section 3 must be enumerated on each node so as to actually perform the computations and communications. We have sketched before some criteria for this enumeration to be efficient. Much work has been devoted to the enumeration problem, with various assumptions about the mapping (block, cyclic, general cyclic distribution, with or without alignment stride), the access (shift, array section or full loop nest) and with various goals (enumerating the local iterations, or generating communication). In this section, where local addresses are concerned, the underlying space is the template of each array: physical addressing will be considered in Section 6. This means that the expected result is scanning δ, ψ and γ variables, that is dealing with the set in the abstract processor, block and cycle space. Three approaches can be identified: *closed forms*, *automata* or *linear integer programming* techniques such as Diophantine equations and polyhedron theory. They are presented in the following, with an emphasis on the assumptions needed by the techniques.

4.1 Closed forms

The approach consists of finding parametric descriptions of the different sets. A closed form is a formula that just needs to be evaluated once missing values are known. It is used at runtime to compute the different sets or to issue the communications and to perform the computations. Many important and practical cases have closed form solutions: For instance, block-distributed arrays and shift accesses communications of the borders is vectorized outside of the loop nest after guard elimination [45] (Figure 24).

The closed form approach generally applies to array sections references. An *array section* is a regular set of integers described by three parameters: the first value, the last value, and an intermediate stride (the default stride value is 1):

$$(l : u : s) = \{i | \exists j \geq 0, i = l + sj, (s > 0 \wedge i \leq u) \vee (s < 0 \wedge i \geq u)\}.$$

Array sections are closed under intersection ($\cap$) and affine transformation ($a.S + b$ where a and b are integers and S is a section).

The reason for the interest in is that they are powerful enough to express the different sets [30] and their computations as outlined in Section 3.4 for array expressions. Also scanning an array section by a `do` loop is trivial. First, the index sets are array sections on block or cyclic distributed dimensions. Figure 13 shows an example. In general, if a one-dimensional array `A` of size n is distributed `block(`k`)` and array `B` is cyclic-distributed onto q processors then with $0 \leq p < q$:

$$\begin{aligned} \mathcal{O}_{\mathtt{A}}(p) &= (pk : \min(pk + k - 1, n)) \\ \mathcal{O}_{\mathtt{B}}(p) &= (p : n : q) \end{aligned}$$

where p remains to be evaluated. Since the only operators used to build the compute and communication sets are the intersection and affine transformation, and since the original sets (owned elements and accessed array sections) are already array sections, the resulting sets are array sections.

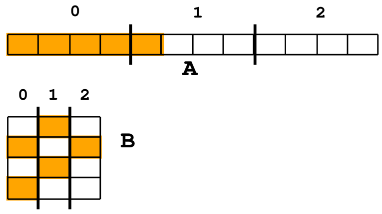
This basic technique was extended [38, 22, 6] to manage general cyclic distributions. A general cyclic distribution is viewed as a block distribution on more *virtual* processors, which are then assigned to *real* processors in a cyclic fashion. This is the *virtual* cyclic view. The dual approach is to consider a cyclic distribution and then to group blocks of *virtual* processors. For alignment strides [27], the sets are computed at the template level and then compressed to remove the holes and generate local addresses.

This simple and parametric technique was quickly introduced in commercial compilers. However it is limited to array section accesses: simple triangular areas cannot be handled. The virtual processor extension results in many intersections to be performed at runtime, for each pair of virtual processors. From a practical point of view, the technique looks like the PARTI system (Section 5), *i.e.* communications are precomputed (on a dimension per dimension, processor per processor and array section basis) and used afterwards.

4.2 Finite State Machines

General cyclic distributions display periodic patterns on the cycle (γ) axis for regular section accesses: the processor and offset may become identical while

```
      array A(0:11)
      array B(0:11)
chpf$ processors P(0:2)
chpf$ distribute A(block) onto P
chpf$ distribute B(cyclic) onto P
      A(0:4) = B(1:9:2)
```



$p =$	0	1	2
$\mathcal{O}_{\mathtt{A}}(p)$	0:3	4:7	8:11
$\mathcal{O}_{\mathtt{B}}(p)$	0:9:3	1:10:3	2:11:3
$\mathcal{C}(p)$	0:3	4	-
$\mathcal{S}_{\mathtt{B}}(0,p)$	3	9	-
$\mathcal{S}_{\mathtt{B}}(1,p)$	1:7:6	-	-
$\mathcal{S}_{\mathtt{B}}(2,p)$	5	-	-

$$
\begin{aligned}
\mathcal{D}_{\mathtt{A}} &= 0:4 \\
\mathcal{D}_{\mathtt{B}} &= 1:9:2 \\
\mathcal{D}_{\mathtt{B}} &= 2\mathcal{D}_{\mathtt{A}}+1 \\
\\
\mathcal{C}(p) &= \mathcal{O}_{\mathtt{A}}(p) \cap \mathcal{D}_{\mathtt{A}} \\
\mathcal{V}_{\mathtt{B}}(p) &= 2\mathcal{C}(p)+1 \\
\\
\mathcal{S}_{\mathtt{B}}(p,p') &= \mathcal{O}_{\mathtt{B}}(p) \cap \mathcal{V}_{\mathtt{B}}(p')
\end{aligned}
$$

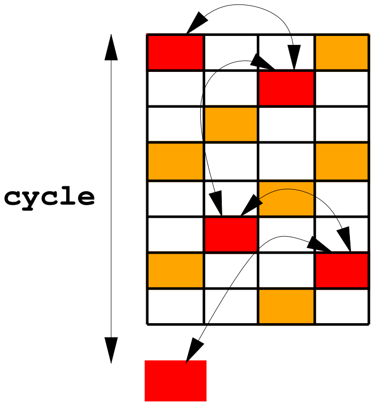
Figure 13: Closed form example: Computation and communication sets

enumerating some sets. For instance, in Figure 11, the access pattern for array section `A(0:42)` repeats each four cycles. Hence, the local enumeration loop has to be unrolled to some extent: for the references corresponding to the first period, tables are precomputed, or divide and remainder computations are performed on the fly; for the following references, the appropriate stride is used. The period depends on the distribution and alignement parameters.

The technique looks for the basic pattern and uses an automaton to enumerate the elements. It was first suggested in [11]. The algorithm builds at runtime the required transition table, which is then used as the base of the scanning. The pattern of the example in Figure 11 is shown in Figure 14. The array elements are light shaded but the accessed ones (array section `A(0:42:3)`) are dark shaded.

The algorithm for building the transition table initially deals with an array section and a general cyclic distribution. The extension to alignment strides simply applies this algorithm twice. The generalization to multidimensional arrays is straightforward. The technique handles simple regular communications, such as shifts. The initial algorithm was improved to generate the transition tables faster [23, 28, 40]. In [29] the method is extended to enumerate affine and coupled subscript accesses for general cyclic distributions. Such extensions typically require several tables per array dimension to handle the various indexes appearing in an affine subscript.

These automaton-based techniques are fast at enumerating local elements. They handle directly the local addresses generation with maximum cache ben-




```
// Delta holds the pattern
// which is computed at runtime
Delta[4] = { 2, 5, 2, 3 } ;

// index in the pattern
i = 0 ;
a = initial ;

// loop to scan the pattern
while (a <= final) {
  // computation with address a
  ... ;

  // next location
  a += Delta[i] ;
  i = (i+1) % 4 ;
}
```

Figure 14: FSM-based enumeration

efits, while other techniques generate higher level code which depends on subsequent optimizations for handling actual addresses efficiently. They are especially targeted to array sections. Extensions are needed to handle general communication sets. The automaton technique solves one of the HPF compilation problems: enumerating sets efficiently. However, these sets (array section on general cyclic distributions) are limited with respect to the description of general communications involving arbitrary HPF mappings. Also the transition tables must be managed at runtime. Another view one can have from the FSM approach is that it allows to accelerate enumeration of complex sets by precomputing the patterns. However, generating the transition tables is as complicated (or simple) as enumerating the required elements directly.

4.3 Polyhedra and lattices

Other techniques rely on parametric sets defined by affine constraints and generate code directly from this representation. Such a representation can be built for any parallel loop with affine access functions and iteration domain. It follows from the affine framework sketched in Section 3.7: systems of linear equalities and inequalities are derived from the HPF array mapping specification, the iteration set and the array accesses. These sets define . The integer points of these polyhedra express the computations and the necessary communications. Enumerating these sets so as to actually perform the computation or generate communications falls under the general category of scanning polyhedra (or projections of polyhedra).

Let us consider the assignment `B=A` in Figure 15. The distributions involve partial replication along some dimension of the source and target processor arrangements. The mapping is depicted in Figure 17. From this code and

```
      real A(20), B(20)
c
c source mapping
c
!hpf$ processors Ps(2,2,2)
!hpf$ template Ts(2,20,2)
!hpf$ align A(i) with Ts(*,i,*)
!hpf$ distribute Ts(block,block,block)
!hpf$.     onto Ps
c
c target mapping
c
!hpf$ processors Pt(5,2)
!hpf$ template Tt(20,2)
!hpf$ align B(i) with Tt(i,*)
!hpf$ distribute Tt(cyclic(2),block)
!hpf$.     onto Pt
c
c assignment:
c
      B = A
```

Figure 15: Distributed array assignment

`A` mapping and addressing (β_1):

$$\begin{array}{l} 1 \leq \alpha_1 \leq 20,\ 1 \leq \theta_1 \leq 2,\ 1 \leq \theta_2 \leq 20, \\ 1 \leq \theta_3 \leq 2,\ 1 \leq \psi_1 \leq 2,\ 1 \leq \psi_2 \leq 2, \\ 1 \leq \psi_3 \leq 2,\ 1 \leq \delta_2 \leq 10, \\ \alpha_1 = \theta_2,\ \theta_2 = 10\psi_2 + \delta_2 - 10,\ \beta_1 = \delta_2 \end{array}$$

`B` mapping (OCR) and addressing:

$$\begin{array}{l} 1 \leq \alpha'_1 \leq 20,\ 1 \leq \theta'_1 \leq 20,\ 1 \leq \theta'_2 \leq 2, \\ 1 \leq \psi'_1 \leq 5,\ 1 \leq \psi'_2 \leq 2,\ 1 \leq \delta'_1 \leq 2, \\ \alpha'_1 = \theta'_1,\ \beta'_1 = 2\gamma'_1 + \delta'_1 - 3, \\ \theta'_1 = 10\gamma'_1 + 5\psi'_1 + \delta'_1 - 15 \end{array}$$

reference and load-balance

$$\alpha_1 = \alpha'_1,\ \psi'_1 = 4\lambda + 2\psi_3 + \psi_1 - 3$$

Figure 16: Full copy system

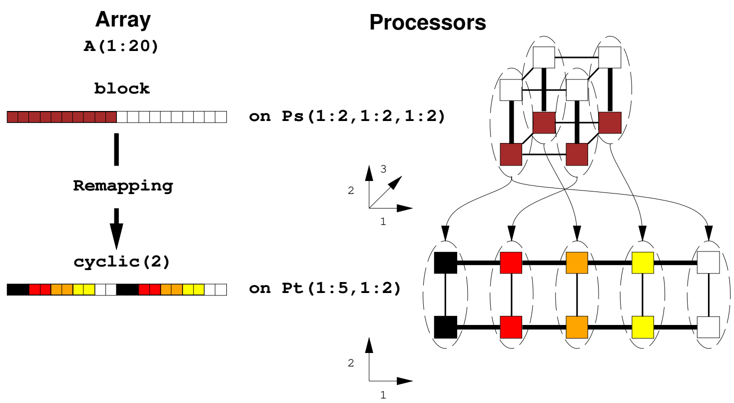


Figure 17: Full copy of distributed arrays

```
IF (I AM IN(Ps)) THEN
  ψ = (ψ_1,ψ_2,ψ_3) = MY ID IN(Ps)
  DO λ = 0, (7-ψ_1-2*ψ_3)/4
    ψ'_1 = -2+ψ_1+2*ψ_3+4*λ
    index = 0
    DO β_1 = 2*ψ'_1-1, 2*ψ'_1
      BUFFER(index++) = A(β_1)
    ENDDO
    CALL BROADCAST(BUFFER,Pt(ψ')) ! send
  ENDDO
ENDIF

IF (I AM IN(Pt)) THEN
  ψ' = (ψ'_1,ψ'_2) = MY ID IN(Pt)
  DO ψ_1 = 1, 2
    DO ψ_3 = 1, 2
      DO λ = (5-ψ_1-2*ψ_3+ψ'_1)/4, (2-ψ_1-2*ψ_3+ψ'_1)/4
        DO ψ_2 = 1, 2
          IF (Ps(ψ)<>Pt(ψ')) THEN ! receive and unpack
            CALL RECEIVE(BUFFER,Ps(ψ))
            index = 0
            DO β'_1 = 2*ψ_2-1, 2*ψ_2
              B(β'_1) = BUFFER(index++)
            ENDDO
          ELSE ! local copy
            DO β_1 = 2*ψ'_1-1, 2*ψ'_1
              β'_1 = β_1+2*ψ_2-2*ψ'_1
              B(β'_1) = A(β_1)
            ENDDO
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDIF
```

Figure 18: Generated SPMD code outline

following the affine framework one can derived the systems in Figure 16. It incorporates the mapping equations, a local addressing scheme (β), the array reference (a simple $\alpha'_1 = \alpha_1$) and an additional equation to express the desired load-balance to benefit from replication. From these equations communication code as presented in Figure 18 may be generated. It scans the solutions to the system so as to generate the required communication and performs the assignment. General examples can be handled with this approach.

For generating such a code, many degree of freedom remain and various approaches have been suggested. The generic sets may be decomposed [43, 17] into linear independent Diophantine equations that are solved using standard methods. Full HPF mappings for array sections [43] and full affine accesses and iteration space for cyclic distributions [17] are considered. The latter paper performs unimodular changes of basis on general affine problems to build simpler array section problems. Moreover, the enumeration scheme is directed by communication vectorization, leading to a particular order of variables in the enumeration scheme. A linear Diophantine equation is:

$$\{(i, j) \in Z^2 | ai = bj + c\}$$

It is related to the intersection of array sections. However, unlike the closed form approach, the decomposition process into these simple equations is systematic.

This decomposition needs not to be performed explicitly [2, 33, 5]. Generic algorithms for automatically generating codes that enumerate solutions to systems of linear equalities and inequalities, based on Fourier elimination [25, 3, 32] or on a parametric simplex [15] are used. These algorithms are potentially exponential, but practical cases are usually handled quickly. The generated code efficiency depends on the ability of the code generation algorithm. Polyhedra are scanned with loop nests involving sometimes complex bounds (min, max, integer divisions) that require special optimizations.

Because these techniques are based on parametric sets, communication and enumeration issues are quite natural; in theory they do not require special attention. However, practice shows that care must be taken to avoid inefficient enumerations. Such techniques clearly address the need for compile-time management of index sets. Some implementations of polyhedron based techniques have been developed for general HPF mappings, for I/O related communications [13] and for remappings [14], with associated optimizations that takes advantage of load balancing and broadcast opportunities.

5 Irregular problems

In order to compile general parallel loops with irregular accesses to distributed variables, it is necessary to rely on the run-time to compute the execution set $\mathcal{C}(p)$ and the communication sets $\mathcal{S}_{\mathtt{B}}(p, p')$.

We describe here how these accesses are handled in the PREPARE compiler [44]. In the PREPARE compiler, all HPF data-parallel constructs are transformed into a standard construct (the ArrayAssign) in the intermediate representation level. The compilation scheme used for an ArrayAssign [8] generates a three-phase code: the **work distributor**, the **inspector**, and the **executor**. The work distributor determines how to spread the work (iterations) among the HPF processors. The inspector analyzes the communication patterns of the loop, computes the description of the communication, and derives translation functions between global and local index spaces, while the executor performs the actual communication and executes the loop iterations. Each phase relies on run-time functions embedded in the PARTI+ library.

5.1 The Parti+ library

The purpose of the PARTI and PARTI+ libraries is to compute non-local accesses at run-time and to perform global exchanges needed to access non local data. The PARTI+ library is an extension of the PARTI library developed at the University of Vienna ; it supports all HPF distribution and alignment types and operations on all Fortran 90 intrinsic data types; moreover, a part of the PARTI kernel was reviewed and an optimized version is implemented in PARTI+. The three main functions of PARTI+ are described below:

5.1.1 cview

The cview routine computes, for a given distributed array, a *schedule* that is used to bring in the non-local data accessed by the calling SPMD process within the body of an ArrayAssign handled by the distributor-inspector-executor technique. The computed schedule is a structure that contains the necessary information to identify the locations in distributed memory from which data is to be accessed. This routine also computes the total number of the distant accesses (offproc) performed by this process, and generates an indirection table (loc) that is used to transform global references into local memory addresses. While constructing the indirection table loc, the cview routine filters out the duplicate references.

5.1.2 gather

The gather routine performs a global exchange between all the calling processes. The data to be exchanged are characterized by the schedule passed as a parameter (this schedule should have been previously computed by the cview routine). At the end of this exchange, all the distant data accessed by a given process within the corresponding parallel loop are locally available to this process: the distant values are stored in the buffer part of the extended local segment. Once these distant elements are received and stored, the array elements can be accessed via the local reference list (loc) that is previously computed by the cview

```
        INTEGER, PARAMETER:: N1 = 4, N2 = 8, M1 = 2, M2 = 4
        REAL, DIMENSION(N1,N2) :: A, B
!HPF$   PROCESSORS P(M1,M2)
!HPF$   DISTRIBUTE (CYCLIC,BLOCK) ONTO P :: A
!HPF$   DISTRIBUTE (BLOCK,CYCLIC) ONTO P :: B
        . . .
        FORALL (i=1:N1 , j=1:N2)
           A(fl(i),gl(j)) = B(fr1(i),gr1(j)) * B(fr2(i),gr2(j))
        END FORALL
```

Figure 19: Example of irregular accesses in an Array Assign statement

routine.

5.1.3 scatter

The scatter routine performs a global scatter exchange between all the calling processes. The data to be exchanged are characterized by the schedule passed as a parameter (this schedule should have been previously computed by the cview routine). At the end of this exchange, all the non-local data written by a given process within the corresponding parallel loop[1] are copied back to their owners (i.e. the processes to which these elements were assigned by the data distribution specification): the distant values were stored in the buffer part of the extended local segment during their computation. The array elements were accessed via the local reference list (loc) that was previously computed by the cview routine.

5.2 Generated code

We describe here the main characteristics of the generated code for the example depicted in Figure 19

5.2.1 Prologue

The generated code begins with the initialization of the mapping descriptors associated with the distributed arrays A and B: the information about local segment size, dimension, shape, distribution and processor array is stored there. Then the local segment of B is allocated.

[1] for which a work distribution that deviates from the *owner-computes rule* has been applied

5.2.2 Work distributor phase

On each SPMD process p, the work distributor computes the *execution set*, $\mathcal{C}(p)$, i.e. the set of loop iterations to be executed by p. In our example, the work distributor applies the *owner-computes rule* (OCR) work distribution strategy, but the PARTI+ library provides features (the scatter routine) that allow us to deviate from this rule whenever another work distribution policy might be better. For the example in Figure 19, the execution set is given by:

$$\mathcal{C}(p) = \{(i,j) \mid (fl(i), gl(j)) \ \in \mathcal{O}_{\mathtt{A}}(p)\}$$

where $\mathcal{O}_{\mathtt{A}}(p)$ denotes the set of indices of elements of A that are stored on processor p.

This set is stored in an array called the *Execution Set Descriptor* (ESD). While filling the ESD, the total number of accesses to B performed by the process is computed and stored.

5.2.3 Inspector phase

The inspector performs a dynamic loop analysis. Its task is to describe the necessary communication by a set of *schedules* that control runtime procedures moving data between processes in the subsequent executor phase. The dynamic loop analysis also establishes an appropriate addressing scheme to access local elements and copies of non-local elements on each process.

The inspector phase starts by the allocation of two integer arrays, glob and loc. The glob array is filled with the value of index expressions of the global references to B performed by the process. Next, the inspecor fills the indirection table loc and evaluates the number of non-local elements of B (offproc) accessed by the calling process. A graphical sketch of how the cview routine works is depicted in Figure 20.

5.2.4 Executor phase

The executor is the third phase generated by the compilation scheme used to transform an irregular ArrayAssign. It performs the communication described by the schedules, and executes the actual computations for all iterations of $\mathcal{C}(p)$ as defined by the work distributor phase. The schedules control communication in such a way that execution of the loop using the local reference list accesses the correct data in local segments and buffers.

The executor phase first checks if the space allocated for the local segment of B is big enough to store the non-local elements; if not, more space for B is allocated. The non-local data are then gathered and stored in the extended local segment thanks to a call to the gather routine. Once the global exchange is done, the iterations assigned to the process by the work distributor are computed. The iteration space of this loop is characterized by the ESD constructed during the

i_1 = esd[0][0], j_1 = esd[0][1], ..., i_{ess} = esd[ess-1][0], j_{ess} = esd[ess-1][1]

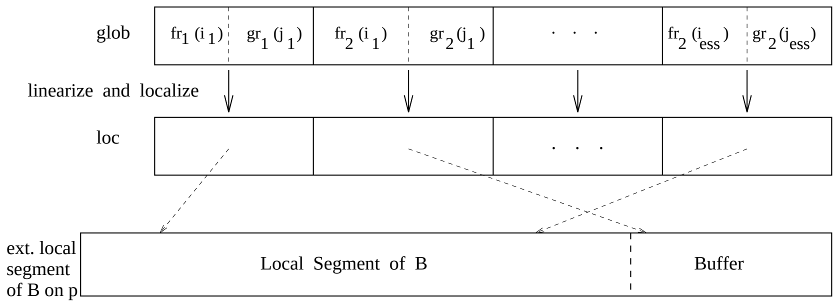


Figure 20: Relation between multi-dim. global references and linearized local references

work distribution phase. All the read-accesses to B are performed through the indirection array loc. The *global-to-local* call, g2l(map_A, x, y), used in the index expression of the LHS, determines the offset of the element A(x,y) in the local segment of A associated with the calling process.

5.2.5 Epilogue

Finally, the intermediate array that stores the local reference list, the schedule and the ESD are freed.

5.3 The ESD and schedule reuse

To make ESD and schedule optimizations possible, the code generation was designed so that it is easy to recognize creation and use of an ESD and a schedule. The generation of PARTI+ calls has been separated into a high-level part (that inserts high-level nodes in the intermediate representation) and a low-level part (that expands the abstraction to loops and routine calls). Between those two parts, a dedicated optimizing engine can be inserted. The Schedule Reuse Analysis Engine (SRAE) performs analysis on these high-level nodes and tries to hoist them outside of loops.

6 Memory allocation and addressing

The compiler must use an allocation and addressing scheme for distributed arrays to switch from the global name space provided to the user by HPF to

```
forall (i ∈ C(p))
    A(f'(i)) = (g1(i) ∈ O_B(p) ? B(g'1(i)) : T1(t1(i))) +
                (g2(i) ∈ O_B(p) ? B(g'2(i)) : T2(t2(i)))
```

$$\texttt{forall}\ (i \in \mathcal{C}(p))\quad \texttt{A}(f'(i)) = (g_1(i) \in \mathcal{O}_{\texttt{B}}(p)\ \texttt{?}\ \texttt{B}(g'_1(i))\ \texttt{:}\ \texttt{T1}(t_1(i)))\ + (g_2(i) \in \mathcal{O}_{\texttt{B}}(p)\ \texttt{?}\ \texttt{B}(g'_2(i))\ \texttt{:}\ \texttt{T2}(t_2(i)))$$

Figure 21: Guarded references

$$\texttt{forall}\ (i \in \mathcal{C}_{\texttt{ll}}(p))\ \texttt{A}(f'(i)) = \texttt{B}(g'_1(i)) + \texttt{B}(g'_2(i))$$
$$\texttt{forall}\ (i \in \mathcal{C}_{\texttt{rl}}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{B}(g'_2(i))$$
$$\texttt{forall}\ (i \in \mathcal{C}_{\texttt{lr}}(p))\ \texttt{A}(f'(i)) = \texttt{B}(g'_1(i)) + \texttt{T2}(t_2(i))$$
$$\texttt{forall}\ (i \in \mathcal{C}_{\texttt{rr}}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{T2}(t_2(i))$$

Figure 22: Compute set splitting

$$\texttt{forall}\ (i \in \mathcal{C}(p))\ \texttt{A}(f'(i)) = \texttt{T1}(t_1(i)) + \texttt{T2}(t_2(i))$$

Figure 23: All in temporary ...

the local name space available on the distributed memory target architecture. This scheme should be driven by the mapping directives.

6.1 Issues

The memory allocation problem is twofold: the compiler must allocate permanent memory storage for the distributed arrays; it must also manage some temporary storage for remote references that are used in the computation process. For instance, in Figure 7, a permanent storage termed x is required for the local blocks of array X, and a temporary storage termed *temp* is used for the neighborhood of the corresponding block in array Y needed by the local computation.

6.1.1 Allocation and addressing trade-off

The following trade-off must be addressed: On the one hand, the amount of memory allocated to store a given array should be as small as possible, because memory is expensive. Moreover the amount of wasted memory for a given dimension is amplified by multidimensional distributions: if one third of each index space on each dimension of a 3D array is wasted, then 70% (19/27) of the allocated memory is wasted on the whole. On the other hand, the chosen scheme must allow fast local address computations, *i.e.* an efficient *global to local* translation, or at least its efficient incremental management within a loop nest.

Most parallel machines HPF was designed for are cache architectures. It is thus interesting to devise the allocation and enumeration schemes together so as to benefit from cache effects.

6.1.2 Temporary storage management

The second issue is the management of the temporary storage for remote references. The same constraints hold, but the memory can be reused. The temporary space needed may vary from one processor to another. Separating the local and temporary allocations can be unexpectedly costly: Let us consider again the generic example in Figure 6, and assume that two separate temporaries are used to store remote *rhs* references. Then for generating the new local computation loop, it must be decided for each reference whether it is local or remote. Guarding all references (Figure 21) is too costly. Moving these guards up by splitting the compute set (Figure 22) seems attractive but the process is exponential with respect to the number of references and is not necessary possible or efficient. Thus the only general technique (Figure 23) beyond overlap allocation which only suits simple cases is to manage local references as the temporary array or *vice versa*, *i.e.* to provide a local allocation scheme which stores remote references homogeneously [34] or to copy the local references into the temporary array [2].

6.2 Techniques

Let us focus on the allocation schemes suggested in the literature. These schemes are outlined for basic examples, and their pros and cons with respect to allocation overhead, fast addressing, and temporary storage are discussed.

Regular compression

```
      real A(10,10), B(10,10)
chpf$ processors P(2,2)
chpf$ align A with B
chpf$ distribute B(block,block) onto P
chpf$ independent...
c
c typical stencil computation
c
      DO j=...
        DO i=...
          B(i,j) =
     .      A(i,j)+A(i-1,j)+A(i,j-1)
        END DO
      END DO
```

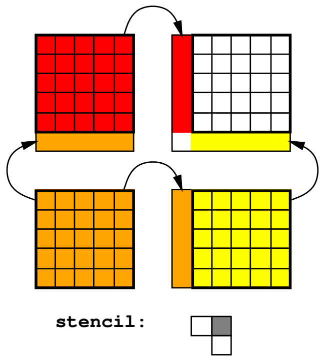
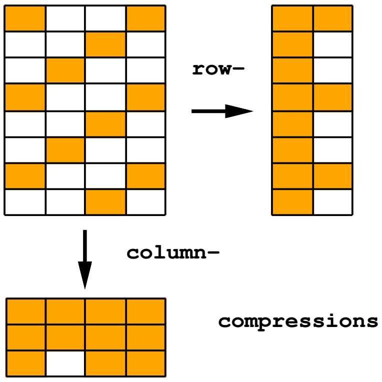


Figure 24: Allocation of overlaps

The first and most natural idea for managing local elements is to adapt the initial dimension by dimension Cartesian allocation in order to remove the regular holes coming from the cyclic distribution and alignment. For block-distributed dimensions, this may be done without any address translation [21].



2-D template view from $\mathtt{A}(\alpha)$:

$$\begin{aligned} \theta &= 3\alpha \\ \gamma &= \theta \div 16 \\ \delta &= \theta \bmod 4 \end{aligned}$$

Row-compression addressing (r, c):

$$\begin{aligned} r &= \delta \div 3 \\ c &= \gamma \end{aligned}$$

Column-compression addressing (r, c):

$$\begin{aligned} r &= \delta \\ c &= \gamma \div 3 \end{aligned}$$

Figure 25: Regular compression for local allocation

For simple shift accesses the technique [18] extends the local allocation to store the remote elements on the border, simplifying addressing. This suits stencil computations very well as shown in Figure 24.

For general cyclic distributions with alignments, both row- and column–compressions are possible [43, 37], as shown in Figure 25. The first one suits large cyclic distributions and the second one small cyclic distributions. The *global to local* functions involve integer divisions and modulus, but this is not an issue: most of the time no actual translation is needed and the *right* local address can be generated directly while enumerating local elements. This scheme is compatible with overlaps and may be improved to reduce the allocated memory but with a higher address computation cost [2].

Packed form

Instead of preserving a structured allocation, it has been suggested to pack local elements as much as possible [11]. This is obviously optimal from the memory allocation point of view. The issue is the addressing problem: accessing the local elements from the base is possible with a finite state machine which computes the relative offsets at runtime and exactly, but this does not correspond necessarily to a simple *global to local* formula. Also the management of such an allocation scheme on multi dimensions is not obvious, because each first element of vectors are not mapped regularly in the memory, since the offset depends on the actual number of elements on the processor for each original vector of the matrix. Managing these issues across procedure boundaries seems also difficult because not much knowledge is available at compile time. This scheme suits only the FSM compilation technique presented in Section 4.2.

Software pagination

```
      real A(0:23)
chpf$ processors P(0:2)
chpf$ distribute A(cyclic(4)) &
chpf$      onto P
```

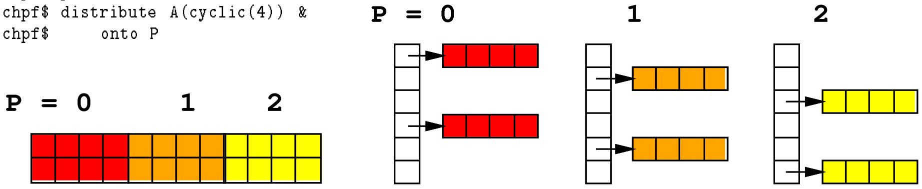


Figure 26: Software paging array management

```
      B(0:23)
chpf$ align B with A
      B(2:23) = A(0:21)+A(2:23)
```

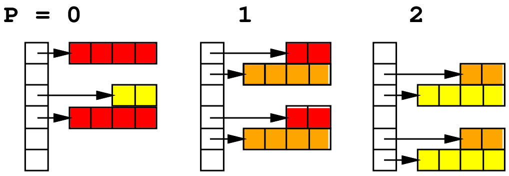


Figure 27: Temporary remote references management

Classical paging systems for memory management can also be used for managing distributed arrays, with both compiler and runtime support [34]. The basic idea is to divide the array space into pages, and then to allocate on a given processor only the pages, or part of pages, on which it owns some array elements. This is depicted in Figure 26. Each array element at address i is associated a page number and offset from the page size s:

$$\begin{array}{rl} \text{page number:} & n = i \bmod s \\ \text{page offset:} & o = i \div s \end{array}$$

The runtime stores a vector of pointers to each page, some of which are allocated. The allocation overhead is small for block or large cyclic distributions, but can be as large as the array itself when considering cyclic distributions. Fast addressing is possible, because the page number and offset computations can use shift and mask operations. Locality and thus cache benefits is preserved on a page per page basis. The main advantage of such an allocation scheme is that it integrates naturally temporary management in an uniform way. Indeed, remote references can be stored within the pages they belong to on each processor, as depicted in Figure 27. Once transfered, the remote elements are stored as the owned array elements and accessed uniformly from the computation loop. The drawbacks are the allocation overhead to be paid for small cyclic distributions, alignment strides and also multidimensional distributions. Also this scheme has some impact on the runtime system, which much manage the allocated pages, and the addressing, although fast, may not be as optimizable as other simpler schemes.

Hash table based allocation and addressing have also been suggested for temporary reference management. The access cost is then uniform, but the random access in memory does not benefit much from caches. These allocation schemes are very difficult to compare one to the other, because they must be considered within the compilation process as a whole, involving enumeration issues. Also the very same scheme may be managed dynamically or statically, compromising subsequent standard optimizations or requiring special optimizations.

7 Conclusion

HPF is a promising language and is the focus of much attention from potential users. It may be a simple and efficient alternative to low level message passing programming. However HPF does not suit all applications. Even for data-parallel applications, a change in data-structures, algorithms and programming habits may be required. Users must understand the compiler technology to get high performance from their code, as they did with vector computers twenty years ago.

From the compiler point of view, the more knowledge available at compile time, the better the optimizations that can be performed and the better the resulting code. This knowledge may be provided in the language, but may also need to be extracted automatically from the program. This analysis may be defeated by bad programming habits, compromising the benefits expected from the parallel version of a program. The language must also allow the user to express his knowledge and advice about the application: some language features such as the `inherit` directive just provide the opposite, no information. Finally compiler technology must catch up with the HPF language, the design of which was beyond the know-how of its time.

We are optimistic that, with effort from users in writing clean data-parallel codes, from the HPF Forum in improving the language design where deficiencies have been detected, and from compiler writers in implementing new compilation techniques, HPF will be viewed as a success story.

Acknowledgment

We are thankful to Robert Schreiber and Monica Lam for their fruitful comments on an early version of this paper.

References

[1] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993.

[2] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, December 1993. New version to appear in *Scientific Programming*, 1996.

[3] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. In *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[4] ANSI. *American National Standard Programming Language Fortran, ANSI x3.9-1978, ISO 1539-1980*. New-York, 1983.

[5] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. An Overview of the PARADIGME Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10), October 1995.

[6] Siegfried Benkner, Peter Brezany, and Hans Zima. Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler. In *5th International Conference on Compiler Construction*, April 1994. Springer-Verlag LNCS vol. 786, pages 324–338.

[7] Vincent Bouchitté, Pierre Boulet, Alain Darte, and Yves Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. *Int. J. of Supercomputer Applications and High Performance Computing*, 9(3):205–219, Fall 1995.

[8] P. Brezany, O. Chéron, K. Sanjari, and E. van Kronijnenburg. Processing irregular codes containing arrays with multidimensional distributions by the PREPARE HPF compiler. In *The International Conference and Exhibition on High-Performance Computing and Networking*, pages 526–531, Milan, Italie, May 1995. LNCS 919, Springer Verlag.

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

[10] Siddharta Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Modeling data parallel programs with the alignment-distribution graph. *Journal of Programming Languages*, (2):227–258, 1994.

[11] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. *Jounal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[12] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, April 1994.

[13] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[14] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. Technical Report A 277, CRI, École des mines de Paris, October 1995. To appear in JPDC, 1996.

[15] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[16] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, November 1994. *Version 1.1*.

[17] Cécile Germain and Franck Delaplace. Automatic Vectorization of Communications for Data-Parallel Programs. In *Euro-Par'95, Stockholm, Sweden*, pages 429–440, August 1995. Springer Verlag, LNCS 966.

[18] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Vienna, 1989.

[19] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *ACM Int. Conf. on Supercomputing*, pages 1–11, 92.

[20] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication, LNCS 892. In *7th Workshop Languages and compilers for parallel computing*, pages 266–282. Springer-Verlag, 94.

[21] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF Compiler for the IBM SP2. In *Supercomputing*, December 1995.

[22] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *International Conference on Parallel Processing*, pages II–301–II–305, August 1993.

[23] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation techniques for block-cyclic distributions. In *ACM International Conference on Supercomputing*, July 1994.

[24] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[25] François Irigoin. Code generation for the hyperplane method and for loop interchange. ENSMP-CAI-88 E102/CAI/I, CRI, École des mines de Paris, October 1988.

[26] ISO/IEC. *International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.

[27] S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling Array Statements for Efficient Execution on Distributed-Memory Machines: Two-level Mappings. In *Language and Compilers for Parallel Computing*, pages 14.1–14.15, August 1995.

[28] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. A Linear Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs. In *Symposium on Principles and Practice of Parallel Programming*, 1995. Sigplan Notices, 30(8):102–111.

[29] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. Efficient address generation for block-cyclic distributions. In *ACM International Conference on Supercomputing*, pages 180–184, June 1995.

[30] Charles Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing*, pages 101–110, November 1991.

[31] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[32] Marc Le Fur. Scanning Parameterized Polyhedron using Fourier-Motzkin Elimination. In *High Performance Computing Symposium*, pages 130–143, July 1995.

[33] Marc Le Fur, Jean-Louis Pazat, and Françoise André. An Array Partitioning Analysis for Parallel Loop Distribution. In *Euro-Par'95, Stockholm, Sweden*, pages 351–364, August 1995. Springer Verlag, LNCS 966.

[34] Yves Mahéo and Jean-Louis Pazat. Distributed Array Management for HPF Compilers. In *High Performance Computing Symposium*, pages 119–129, July 1995.

[35] Edwin M. Paalvast, Henk J. Sips, and A.J. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *1991 International Conference on Parallel Processing — Volume II : Software*, pages 124–131, June 1991.

[36] Alexis Platonoff. Automatic data distribution for massively parallel computers. In *Workshop on Compilers for Parallel Computers, Malaga*, pages 555–570, June 1995.

[37] Henk J. Sips, Kees van Reeuwijk, and Will Denissen. Analysis of local enumeration and storage schemes in HPF. In *ACM International Conference on Supercomputing*, May 1996.

[38] J. Stichnoth, O'Hallaron D., and T. Gross. Generating communication for array statements: Design, implementation and evaluation. *Jounal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[39] E. Su, D.J. Palermo, and P. Banerjee. Automatic parallelization of regular computations for distributed memory multicomputers in the paradigm compiler. In *1993 Int. Conf. on parallel processing*, pages II30–II38. CRC Press, 93.

[40] Ashwath Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs using Integer Lattices. In *Language and Compilers for Parallel Computing*, pages 13.1–13.19, August 1995.

[41] C-W. Tseng. *An optimizing Fortran D compiler for MIMD Distributed- memory machines*. PhD thesis, Rice University, 93.

[42] Vincent Van Dongen. Compiling distributed loops onto SPMD code. *Parallel Processing Letters*, 4(3):301–312, March 1994.

[43] C. van Reuuwijk, H. J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. Computational Physics Report Series, CP-95 006, Delft University of Technology, November 1994.

[44] A. Veen and de Lange M. Overview of the prepare project. In *4th Int. Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993.

[45] Hans Zima and Barbara Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, February 1993.