Optimal Compilation of HPF Remappings (Extended Abstract)

Fabien COELHO
Corinne ANCOURT *

Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau Cedex, FRANCE.
Phone: +33 1 64 69 47 08. Fax: + 33 1 64 69 47 09.
URL: http://www.cri.ensmp.fr/pips

October 23, 1995

Abstract

Applications with varying array access patterns require to dynamically change array mappings on distributed-memory parallel machines. HPF (High Performance Fortran) provides such remappings, on data that can be replicated, explicitly through the `realign` and `redistribute` directives and implicitly at procedure calls and returns. However such features are left out of the HPF subset or of the currently discussed HPF kernel for efficiency reasons. This paper presents a new compilation technique to handle HPF remappings for message-passing parallel architectures. The first phase is global and removes all useless remappings that appear naturally in procedures. The code generated by the second phase takes advantage of replications to shorten the remapping time. It is proved optimal: A minimal number of messages, containing only the required data, is sent over the network. The technique is fully implemented in HPFC, our prototype HPF compiler. Experiments were performed on a DEC Alpha farm.

Keywords:

HPF compilation, array remappings, redistributions, distributed-memory MIMD architecture, message-passing, PVM, static load-balancing, linear algebra, polyhedrons, constraint-based code generation.

*{coelho,ancourt}@cri.ensmp.fr

Contents

Introduction **4**

1 Remapping Graph **5**
1.1 Definition and construction . . . 6
1.2 Optimization . . . 7
1.3 Discussion . . . 8

2 Example and notations **9**

3 Linear formalization **11**
3.1 HPF modelization . . . 12
3.2 Broadcasts and load balance . . . 13

4 SPMD code generation **15**

5 Optimality and discussion **17**

6 Experiments **18**
6.1 Experimental conditions . . . 18
6.2 Performance analysis . . . 21

Conclusion **22**

References **24**

List of Figures

1 Simple ADI-like structured example . . . 5
2 A simple vertex . . . 6
3 G_R for `remaps` at construction start . . . 6
4 Initial G_R for `remaps` . . . 7
5 Optimized G_R for `remaps` . . . 8
6 Running example for code generation . . . 10
7 Array `A` remapping . . . 10
8 declaration constraints $D(\alpha, \theta[\,'], \psi[\,'])$ . . . 12
9 HPF-related constraints $H(\alpha, \theta[\,'], \psi[\,'], \gamma[\,'], \delta[\,'])$ and dimension sets . . . 12
10 local declaration constraints $L(\beta[\,'], \delta[\,'], \gamma[\,'], \alpha)$ . . . 13
11 SPMD remapping code . . . 16
12 Remapping-based transpose code for HPFC . . . 20
13 Transposition speed per processor on `P(2,2)` . . . 21
14 Transposition speed per processor on `P(2,3)` . . . 22
15 Transposition speed per processor for (block,block) distributions . . . 23

List of Tables

1 Variables . . . 11
2 Polyhedrons . . . 11
3 Polyhedron operators . . . 12
4 B target to source assignment for the running example . . . 14
5 Transposition time (seconds) on `P(2,2)` . . . 19
6 Transposition time (seconds) on `P(2,3)` . . . 19
7 Transposition time (seconds) for (block,block) distributions . . . 19

Introduction

Many applications, such as ADI (Alternating Direction Integration) and FFT [18] (Fast Fourier Transform), require different array mappings at different computation phases for efficient execution on distributed-memory parallel machines (*e.g.* CRAY T3D, IBM SP2, DEC Alpha farm). Data replication, sometimes partial, is used to share data between processors. Data remapping and replication often need to be combined: A parallel matrix multiplication accesses a whole row and column of data to compute each single target element, hence the need to remap data with some replication for parallel execution. Moreover, automatic data layout tools [24, 7] suggest data remappings between computation phases. Thus handling data remappings efficiently is an important issue for high performance computing.

HPF (High Performance Fortran [14, 27], a Fortran 90-based data-parallel language) targets distributed-memory parallel architectures. Standard directives are provided to specify array mappings that may involve some replication. These mappings are changed dynamically, explicitly with *executable* directives (`realign`, `redistribute`) and implicitly at procedure calls and returns for *prescriptive* argument mappings. These useful features are perceived as difficult to compile efficiently and thus are left out of the HPF subset or of the currently discussed HPF kernel [15]. If not supported, or even not well supported, applications requiring them will not be ported to HPF...The key issues to be addressed are the reduction of the runtime overheads induced by remappings, and the management of the rich variety of HPF mappings.

Related work

Any technique that handles all HPF array assignments can be used to compile remappings: the induced communications are those of an array assignment `A=B`, where `B` is mapped as the source and `A` as the target. Such techniques are based on finite state machines [6, 20, 25], closed forms [17, 31, 19], diophantine equations [28, 5, 36] or polyhedra [1, 3, 34, 32]. However none of these techniques considers load-balancing and broadcasts. Also issues such as handling different processor sets, multidimensional distributions, communication generation and local addresses... are not all clearly and efficiently managed in these papers, therefore dedicated optimized techniques are needed.

In [33], support by runtime library is suggested for simple cases involving neither shape changing[1], nor alignment or replication. Multidimensional remappings are decomposed into 1-D remappings, hence resulting in several remappings at runtime. *Ad hoc* descriptors called *pitfalls* are devised in [30], but alignment, replication and shape changing are not considered either. A polyhedron-based approach is outlined in [35], for realignments with a fixed general cyclic distribution onto a 1-D processor array. The alignments, unlike HPF, involve arbitrary affine functions.

Contributions

Remapping overheads are attacked at different levels by the compilation technique implemented in HPFC, our prototype HPF compiler. At a global level, all useless remappings are removed. This optimization is presented in the first part of the paper. Such remappings arise naturally in programs.

The second part of paper focuses on the remapping code generation problem for message-passing parallel architectures with non-blocking sends and blocking receives. The problem is fitted into a single powerful linear framework, which integrates all issues. Arbitrary remappings, involving partial replication, alignment strides, general cyclic distributions and differently shaped processor grids are handled. The SPMD generated code is based on the enumeration of polyhedron solutions that abstracts the required communications. Load balancing and broadcasts are also considered. Correctness and optimality results are discussed. The technique is fully implemented in HPFC [8, 9, 10], a prototype HPF compiler developed within the PIPS

[1] The distributed dimensions are the same for both source and target mappings.

project [22]. Experiments on a DEC Alpha farm are also presented. To our knowledge, this technique is the first to integrate all HPF mapping issues in a single framework, to address load balancing and possible broadcasts in the generated code and to present optimality results.

Section 1 presents the remapping graph construction from the control flow graph and a global optimization on this graph to remove statically all useless remappings. Section 2 introduces an example and notations for the code generation. The remapping problem formalization into a polyhedron is described and illustrated in Section 3: The HPF constraints are presented, then optimizations taking into account (1) particular distribution of data onto the processors such as replication and (2) efficient communication capabilities of distributed memory machines such as broadcast, are added to the compilation scheme. The SPMD code generation is presented in Section 4 and optimality properties are discussed in Section 5. Finally, Section 6 presents and analyzes experimental results.

1 Remapping Graph

Useless remappings may appear naturally in HPF programs. First, the change of both alignment and distribution of an array requires a `realign` and a `redistribute`, hence resulting in two remappings if no special care is taken. Second, the redistribution of a template[2] induces the remapping of all aligned arrays, even if they are not all referenced afterwards. Third, at an interprocedural level, two consecutive subroutine calls may require the same remapping for a given array, resulting in a useless remapping on return from the first subroutine and on entry in the second. If two different mappings are required, it may also be interesting to remap data directly rather than using the intermediate original mapping. Such examples do not arise from badly written programs, but from a normal use of HPF features. They demonstrate the need for compile time optimizations to avoid useless costly remappings at runtime.

```
      SUBROUTINE remaps(A)
! distribute A... => A mapping: 0
      local arrays B, C
! template T, align B, C with T, distribute T... => B and C mappings: 0
      use C
      use B
1     redistribute T => B and C mappings: 1
      DO ...
2        remap A... => A mapping 1
         use A
3        remap A... => A mapping 2
         use A
      ENDDO
      use B
      END
```

Figure 1: Simple ADI-like structured example

Let us consider example `remaps` in Figure 1. The loop nest involving two remappings is typical of ADI computations. Template `T` is redistributed at `1`, inducing `B` and `C` remappings, but `C` is not referenced afterwards. Moreover argument `A` is never referenced with its initial mapping.

In this section, the remapping graph, its construction from the control flow graph and its optimizations are presented. This approach deals with descriptive and prescriptive mappings, *i.e.* when the compiler is aware of data distributions.

[2] Even when no templates are used [37] array redistributions generate the problem

1.1 Definition and construction

Let us introduce the remapping graph $\mathcal{G}_R$. This graph is a (usually much smaller) subgraph of the control flow graph. The vertices of the graph are the (re)mapping statements, *i.e.* `(re)align`s and `(re)distribute`s. An edge denotes a possible path in the control flow graph where a same array is remapped at both vertices. Two vertices are added at entry in and on exit from the subroutine. Mappings are designed by a number for each array. The same mapping number is used for identical mappings of an array. To each vertex v in $\mathcal{G}_R$ are associated:

- the set of remapped arrays $\mathrm{S}(v)$
- for all remapped arrays `A` in this set:
  - one leaving mapping[3] for the statement: $\mathrm{L}_{\mathtt{A}}(v)$
  - the set of mappings for `A` that may reach v: $\mathrm{R}_{\mathtt{A}}(v)$
  - whether it may be referenced after the remapping: $\mathrm{Used}_{\mathtt{A}}(v)$

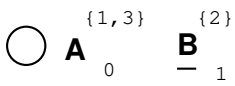

Figure 2: A simple vertex

This information is depicted in Figure 2. To the vertex is associated the remapped arrays `A` and `B`, with the leaving mapping as a subscript (0 for `A`, 1 for `B`) and the set of reaching mappings as a superscript ($\{1,3\}$ for `A`, $\{2\}$ for `B`). Referenced arrays are underlined (here only `B`). The compiler must generate remapping codes for each pair (reaching to leaving mappings).

Let us describe how $\mathcal{G}_R$ is built from the program control flow graph. First, the entry and exit vertices are created. The distributed subroutine local variables are attached to the entry, with their initial mapping as a leaving mapping. The subroutine distributed formal parameters are attached to both entry and exit vertices. The leaving mapping for those variables on entry is the initial mapping. The reaching (resp. leaving) mapping for the entry (resp. exit) is an *a priori* unknown `X` mapping. If the directives are descriptive, the reaching mapping on entry of the subroutine is the initial mapping (`X=0`). On exit, distributed formal parameters are tagged as used: without further interprocedural information, the compiler assumes that the final remapping is needed. Figure 3 shows the initial graph for `remaps`.

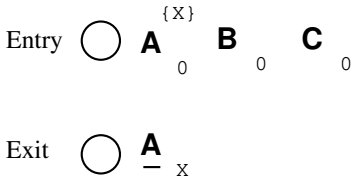

Figure 3: $\mathcal{G}_R$ for `remaps` at construction start

The next phase of the $\mathcal{G}_R$ construction is the propagation of the initial mappings from the subroutine entry, till remappings are encountered. This must be done for each couple $(v, \mathtt{A})$ of vertex and arrays remapped at this vertex. First, initialize the set of couples to be propagated with the entry vertex associated to the distributed arrays. Then for each such couple $(v, \mathtt{A})$,

[3] several may occur, this assumption just simplifies the presentation

propagate in the control graph from the corresponding vertex till meeting remapping statements for that array mapping. Tag the array remapping as used if a reference is encountered while propagating. Let w be one of the encountered remapping statements. Add A to $S(w)$ and compute $L_A(w)$ if necessary, and (w, A) is a new couple to be explored later on. Add $L_A(v)$ to $R_A(w)$. The resulting remapping graph for `remaps` is shown in Figure 4.

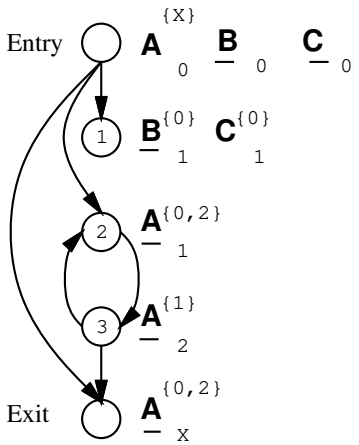

Figure 4: Initial $\mathcal{G}_R$ for `remaps`

If n is the number of vertices in the control graph, s the maximum number of successors of a vertex, m the number of remapping statements and p the number of distributed arrays, then the worst case complexity of the outlined construction algorithm is $\mathcal{O}(nsmp)$, if all arrays are remapped at each remapping statements and the propagations in the control graph get through all vertices.

1.2 Optimization

In $\mathcal{G}_R$, arrays that are remapped after a remapping without having been referenced are tagged as not used for this remapping. In such cases, at least two remappings will be performed at runtime without referencing the array in between, as array A in Figure 4 after the entry vertex. Such useless remappings must be removed. However the successive remapping statements must be aware that they were not performed and that they may have to deal with other reaching mappings. Indeed, the whole set of reaching mappings must be recomputed. Some are no longer of use and others must be added. This optimization is performed as follow:

- First, remove all useless remappings[4], simply by deleting the leaving mapping for those vertices and arrays.

$$\forall v, \forall A \in S(v), \text{not Used}_A(v) \Rightarrow L_A(v) = \emptyset$$

- Second, recompute the mappings that may reach each vertex. This is a forward may data flow problem [26, 23] on $\mathcal{G}_R$:

[4] As a convention in the interpretation of the remapping graph, remappings at vertex v for array $A \in S(v)$ will not be generated if $L_A(v) = \emptyset$.

– initialization: Used 1-step reaching mappings

$$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \mathrm{R}_{\mathtt{A}}(v) = \bigcup_{\substack{w \in \mathrm{pred}(v) \\ \mathtt{A} \in \mathrm{S}(w),\ \mathrm{Used}_{\mathtt{A}}(w)}} \mathrm{L}_{\mathtt{A}}(w)$$

– optimizing function: propagation

$$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \mathrm{R}_{\mathtt{A}}(v) = \mathrm{R}_{\mathtt{A}}(v) \cup \bigcup_{\substack{w \in \mathrm{pred}(v) \\ \mathtt{A} \in \mathrm{S}(w),\ \text{not } \mathrm{Used}_{\mathtt{A}}(w)}} \mathrm{R}_{\mathtt{A}}(w)$$

The iterative resolution of the optimizing function is increasing and bounded, thus it converges. The resulting graph for `remaps` is shown in Figure 5.

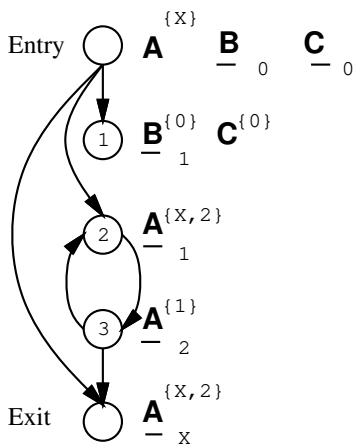

Figure 5: Optimized $\mathcal{G}_R$ for `remaps`

Let us assume a $\mathcal{O}(1)$ set's element put, get and in-test implementation. Let m be the number of vertices in $\mathcal{G}_R$, p the number of distributed arrays, q the maximum number of different mappings for an array and r the maximum number of predecessors for a vertex. Then the worst case time complexity of the optimization, for a simple iterative implementation, is $\mathcal{O}(m^2pqr)$.

This optimization is correct and the result is optimal:

Theorem 1 *The computed remappings (from new reaching to remaining leaving) are those and only those that are needed (according to the static information provided by the data flow graph):*
$\forall v,\ \forall A \in S(v) \wedge Used_A(v),\ \forall a \in R_A(v)$,
$\exists w$ *and a path from* w *to* v *in* $\mathcal{G}_R$*, so that* $a \in L_A(w)$ *and* A *is not used on the path.*

Proof: construction of the path by induction on the solution of the data flow problem. Note that the path in $\mathcal{G}_R$ reflects an underlying path in the control flow graph with no use and no remapping of the array.

1.3 Discussion

- If subroutine local arrays are not used from the entry point in their initial mapping, the compiler may delay the allocation till a used mapping is needed, or chose another initial mapping among directly useful ones.

- Remappings involving unknown `X` mappings should be propagated to call sites in order to be instantiated.

- The set of needed remappings after this optimization may have been reduced or extended. What is minimized is the number of remappings performed at run-time, not those that must be addressed at compile time. Our compiler keeps a database of generated remapping codes in order not to generate some code twice.

- The remapping graph was presented at an intraprocedural level. It is natural to extend it to the interprocedural level, for instance by providing a summary of the entry and exit remappings to be used at the call sites foroptimizations. Remappings of arguments should be decided and performed at call site.

- $\mathcal{G}_R$ for `remaps` includes an edge from the entry to the exit vertex, because the `DO` loop may be empty and thus array `A` may reach the exit vertex without remapping. If the compiler can determine that the loop body is always executed, the skipping edge can be removed from the control graph, thus improving remapping graph $\mathcal{G}_R$ quality.

- In order to simplify the presentation, it was assumed that only one mapping for an array could leave a remapping statement. This is not necessarily the case in HPF. Thus several leaving mappings may be associated to a vertex and array, and for each of these mappings a set of reaching mappings. Care must also be taken in the building phase. Use-information must be attached to the leaving mappings.

- The runtime needs to keep track of the mapping status of each array to chose the right remapping routine when needed.

- Some additional benefits may be obtained by moving remapping in the control flow graph, in order to perform a remapping only when the array is to be actually referenced in its new shape.

2 Example and notations

Let us consider the example in Figure 6. This example is deliberately contrived, and designed to show all the capabilities of our algorithm. Real application remappings should not present all these difficulties at once, but they should frequently include some of them. Vector `A` is remapped from a block distribution onto 3-D processor grid `Ps` to a general cyclic distribution onto 2-D processor grid `Pt` through template `T` redistribution. Both source and target mappings involve partial replication. The corresponding data layouts are depicted in Figure 7. The colors denote the data to processor affectation. The initial mapping is a `block` distribution of `A` onto the second dimension of `Ps`. Each column of (dark and light) processors in `Ps` owns a full copy of `A`. Thus `A` is replicated 4 times. The target mapping is a `cyclic(2)` distribution onto `Pt` first dimension. Each line owns a full copy of `A` and `A` is replicated twice.

Let us describe how the SPMD generated code handles the remapping communications. The arrows in Figure 7 denote the source to target processor assignment. On the target side, each column of processors waits for *exactly* the same data, hence the opportunity to broadcast the same messages to these pairs. On the source side, each column can provide any needed data, since it owns a full copy of `A`. The different columns can deal with different target processors, thus balancing the load of generating and sending the messages. For the running example, 5 different target processor sets are waiting for data that can be addressed by 4 source processor groups. The source to target processor assignment statically cyclically balances the targets among the possible senders.

Linear algebra provides a powerful framework to characterize array element sets, represent HPF directives, generate efficient code, define and introduce optimizations and make possible

```
      parameter (n=20)
      array A(1:n)
chpf$ template T(1:n,1:n,1:n)
chpf$ dynamic A, T
chpf$ align A(i) with T(*,i,*)
c
c Source
c
chpf$ processors Ps(1:2,1:2,1:2)
chpf$ distribute T(block,block,block) onto Ps
c
c Target
c
chpf$ processors Pt(1:5,1:2)
      ...
c
c Array A remapping: Ps(*,block,*) -> Pt(cyclic(2),*)
c
chpf$ redistribute T(*,cyclic(2),block) onto Pt
      ...
```

Figure 6: Running example for code generation

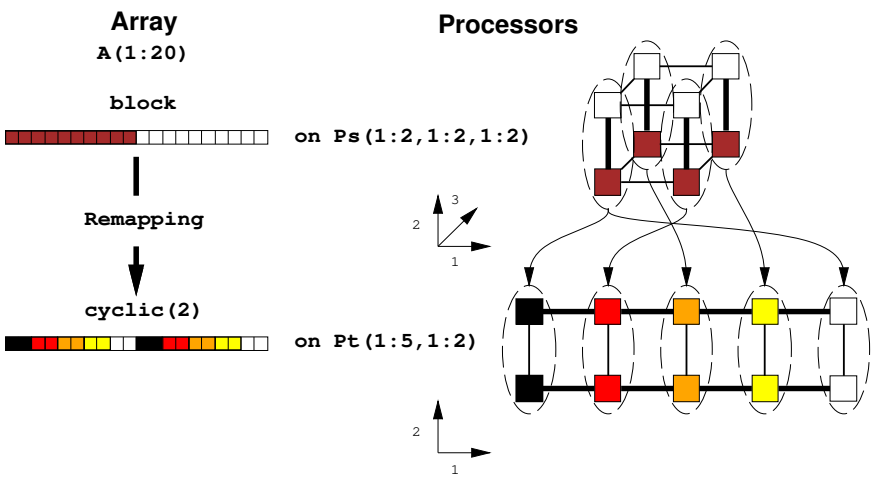

Figure 7: Array `A` remapping

correctness proof of the compilation scheme. Our compilation scheme uses polyhedra to represent the HPF data remapping problem. Notations are shown in Tables 1, 2 and 3. Greek letters denote individual or set of integer variables; calligraphic letters systems of linear equalities and inequalities on a set of variables. Such constraints implicitly define a polyhedron on the variables, *i.e.* the set of integer vectors that are solutions to the system. Different operations can be performed on systems of constraints such as projecting variables, or enumerating solutions for some variables, the others being considered as parameters...

Variables	Comments
α	array dimensions
β	local array dimensions
θ	template dimensions
ψ	processor dimensions
δ	block offsets (for distributions)
γ	cycle numbers (for distributions)
p	all processor variables ($p = \psi \cup \psi'$)
λ	cycle load-balancing variable
e	other variables
ψ_D	set of distributed dimensions
ψ_R	set of replicated dimensions
$\|x\|$	cardinal (or extent) operator
x_i	ith dimension of x
x'	corresponding target mapping variables
$x[']$	shorthand for x and x'

Table 1: Variables

Polyhedrons	Constraints
$\mathcal{D}$	declarations
$\mathcal{H}$	HPF-related
$\mathcal{L}$	local declarations
$\mathcal{B}$	load-balancing
$\mathcal{R}$	remapping
$\mathcal{E}$	elements
$\mathcal{P}$	processors

Table 2: Polyhedrons

3 Linear formalization

Declarations, HPF directives and local address translations are embedded into linear constraints, as suggested in [3, 10]. This gives a linear description of the data distribution and of the communication problem, *i.e.* the enumeration of the elements to be sent and received. This section presents the derivation of a system of linear constraints that exactly describes the array elements to be communicated, with their associated source and target processors, allowing code generation.

Notation	Meaning
$\mathcal{X}(V)$	linear system on variables V
$\mathcal{X}_{\|V}$	system after projection of V variables
$Z \in \mathcal{X}[W]$	$\mathcal{X}$ solution enumeration parametrized by W
$\mathcal{X}_3 = \mathcal{X}_1 \cup \mathcal{X}_2$	union of systems
	(*i.e.* intersection of polyhedrons...)
$\mathcal{X}_3 = \mathcal{X}_1 \times \mathcal{X}_2$	disjoined union
	(*i.e.* union on disjoined set of variables)

Table 3: Polyhedron operators

3.1 HPF modelization

Figure 8 shows the declaration constraints for the objects involved in the source and target mappings of the running example. Lower and upper bounds are defined for each dimension of arrays, templates and processor grids. Figure 9 shows the constraints derived from HPF directives. The sets of distributed and replicated dimensions are also shown. The alignment is quite simple here, but affine expressions are needed for general alignments (for instance `align A(i,*) with T(*,3*i-2)` would lead to $\theta_2 = 3\alpha_1 - 2$). The template distributions require additional variables, for modeling blocks and cycles: δ is the offset within a block; γ is the cycle number, *i.e.* the number of wraps around the processors for cyclic distributions. General cyclic distributions need both variables. Distributions on replicated dimensions are useless, thus are not included. This linear modelization is extensivelly described in [3]. Figure 10 presents the local declarations and global to local address translations generated by HPFC, expressed through linear constraints. Thus they are directly included in our compilation scheme. However such an integration is not required: providing *global to local* address translation functions would be sufficient, although more expensive at run time.

$$
\begin{array}{rl}
\texttt{A}(\alpha_1) & 1 \leq \alpha_1 \leq 20 \\
\text{source template } \texttt{T}(\theta_1,\theta_2,\theta_3) & 1 \leq \theta_1 \leq 20,\ \ 1 \leq \theta_2 \leq 20,\ \ 1 \leq \theta_3 \leq 20 \\
\text{source processors } \texttt{Ps}(\psi_1,\psi_2,\psi_3) & 1 \leq \psi_1 \leq 2,\ \ 1 \leq \psi_2 \leq 2,\ \ 1 \leq \psi_3 \leq 2 \\
\text{target template } \texttt{T}(\theta'_1,\theta'_2,\theta'_3) & 1 \leq \theta'_1 \leq 20,\ \ 1 \leq \theta'_2 \leq 20,\ \ 1 \leq \theta'_3 \leq 20 \\
\text{target processors } \texttt{Pt}(\psi'_1,\psi'_2) & 1 \leq \psi'_1 \leq 5,\ \ 1 \leq \psi'_2 \leq 2
\end{array}
$$

Figure 8: declaration constraints $\mathcal{D}(\alpha, \theta[\,'], \psi[\,'])$

$$
\begin{array}{rl}
\texttt{align A(i) with T(*,i,*)} & \theta_2 = \alpha_1 \\
\textit{idem for target mapping} & \theta'_2 = \alpha_1 \\
\textit{distribution of } \texttt{A} \textit{ onto } \texttt{Ps} \textit{ thru } \texttt{T} & \theta_2 = 10\psi_2 + \delta_2 - 9,\ \ 0 \leq \delta_2 < 10 \\
\textit{distribution of } \texttt{A} \textit{ onto } \texttt{Pt} \textit{ thru } \texttt{T} & \theta'_2 = 10\gamma'_1 + 2\psi'_1 + \delta'_1 - 1,\ \ 0 \leq \delta'_1 < 2
\end{array}
$$

$$
\begin{array}{ll}
\psi_D = \{\psi_2\} & \psi_R = \{\psi_1, \psi_3\} \\
\psi'_D = \{\psi'_1\} & \psi'_R = \{\psi'_2\}
\end{array}
$$

Figure 9: HPF-related constraints $\mathcal{H}(\alpha, \theta[\,'], \psi[\,'], \gamma[\,'], \delta[\,'])$ and dimension sets

$$\begin{array}{ll} \text{local source array } \mathtt{As}(\beta_1) & \beta_1 = \delta_2 + 1,\ 1 \leq \beta_1 \leq 10 \\ \text{local target array } \mathtt{At}(\beta'_1) & \beta'_1 = 2\gamma'_1 + \delta'_1 + 1,\ 1 \leq \beta'_1 \leq 4,\ \gamma'_1 \geq 0 \end{array}$$

Figure 10: local declaration constraints $\mathcal{L}(\beta['], \delta['], \gamma['], \alpha)$

Let us now gather all these constraints in the remapping system (Definition 1). They define a polyhedron on α, β, ψ, $\delta\ldots$ and corresponding primed variables[5]. Solutions to this polyhedron link the array elements α and their mapping on the source ψ and target ψ' processors. $\mathcal{R}$ satisfies some properties because of the HPF mapping semantics.

Definition 1 (Remapping System $\mathcal{R}$)

$$\mathcal{R}(p, e) = \mathcal{R}(\psi['], \alpha, \beta['], \ldots) = \mathcal{D}(\ldots) \cup \mathcal{H}(\ldots) \cup \mathcal{L}(\ldots)$$

with $p = \psi \cup \psi'$ the source and target processor variables, $e = \alpha \cup \ldots$ the other variables.

Proposition 1 (Replication Independence) *Processor variables on replicated dimensions are disjoined from others in $\mathcal{R}$ with $p = p_R \cup p_D$:*

$$\mathcal{R}(p, e) = \mathcal{R}_{|p_R}(p_D, e) \times \mathcal{D}(p_R) = \mathcal{R}_{|p_R}(p_D, e) \times \mathcal{D}(\psi_R) \times \mathcal{D}(\psi'_R)$$

Proof: p_R variables appear neither in $\mathcal{H}$ nor in $\mathcal{L}$, and are disjoined in $\mathcal{D}$. $\mathcal{D}(x)$ is simply the cartesian declaration constraints on x variables. □

Proposition 2 (Disjoined Distribution) *Array elements appear once in $\mathcal{R}_{|p_R}$:*

$$\forall \alpha \in \mathcal{D}(\alpha), \exists!(e, p_D) \text{ with } e = \alpha \cup \ldots | (e, p_D) \in \mathcal{R}_{|p_R}(p_D, e)$$

i.e. *apart from replicated dimensions, only one processor owns a data on the source and target processor grids, thus constraining the possible communications.*

Proof: HPF mapping semantics. □

3.2 Broadcasts and load balance

A SPMD code must be generated from such a polyhedron linking the array elements to their corresponding source and target processors. However, in the general case, because of data replication, $\mathcal{R}$ is not constrained enough for attributing *one* source to a target processor for a given needed array element. Indeed, $\mathcal{R}_{|p_R}$ assigns exactly one source to a target as shown in Proposition 2, but p_R variables are still free (Proposition 1). The underconstrained system allows choices to be made in the code generation. On the target side, replication provides an opportunity for broadcasts. On the source side, it allows to balance the load of generating and sending the messages.

Broadcasts

In the target processor grid, different processors on the replicated dimensions own the same data set. Thus they must somehow receive the same data. Let us decide that the very same messages will be broadcasted to replicated target processors from the source processors. From the communication point of view, replicated target processors are seen as *one abstract* processor to be sent a message. On the polyhedron point of view, ψ'_R dimensions are collapsed for message

[5] Some variables, as θ, are of no interest for the code generation and can be exactly eliminated, reducing the size of the system without loss of generality nor precision.

generation. The free choice on ψ'_R variables is removed, since the decision implies that the source processor choice is independent of these variables!

For the running example, $\psi'_R = \{\psi'_2\}$ and $\mathcal{D}(\psi'_2) = \{1 \leq \psi'_2 \leq 2\}$, thus messages are broadcasted on `Pt`'s second dimension as shown in Figure 7.

Load balancing

Now one sender among the possible ones (ψ_R) must be chosen, as suggested in Figure 7. This choice must be independent of the replicated target processors, because of the broadcast decision. Moreover, in order to minimize the number of messages by sending elements in batches, it should not depend on the array element to be communicated. Thus the only possible action is to link the *abstract* target processors ψ'_D to ψ_R. These processors wait for disjoined data sets (Proposition 2) that can be provided by any source replicated processors (Proposition 1).

To assign ψ'_D to ψ_R in a balanced way, the basic idea is to attribute cyclically distributed target to replicated source processor dimensions. This cyclic distribution must involve processors seen as vectors on both side. In order to obtain this view of ψ'_D and ψ_R, a linearization is required to associate a single identifier to a set of indices.

The rationale for the linearization is to get rid of the dimension structuration in order to balance the cyclic distribution from all available source replicated processors onto all target distributed processors. Source processors that own the same elements are attributed a unique identifier through $\text{lin}(\psi_R)$, as well as target processors requiring different elements through $\text{lin}(\psi'_D)$.

Definition 2 (Linearization) *Let V be a set of bounded variables. The linearization function lin is defined recursively as: $lin(\emptyset) = 0$ and $lin(V) = |v| \cdot lin\ (V - \{v\}) + v - \min(v)$.*

Cardinal operator $||$ is extended to linearized sets with $|lin(\emptyset)| = 1$ and $|lin(V)| = \prod_{v \in V} |v|$.

The following constraint expresses the cyclic distribution of distributed target to replicated source processor dimensions. It introduces a new cycle number variable λ.

Definition 3 (Load Balance $\mathcal{B}$)

$$\mathcal{B}(\psi_R, \psi'_D, \lambda) = \{lin(\psi'_D) = |lin(\psi_R)| \cdot \lambda + lin(\psi_R)\}$$

For the running example, linearization of $\psi_R = \{\psi_1, \psi_3\}$ where $1 \leq \psi_1 \leq 2$, $1 \leq \psi_3 \leq 2$ leads to $\text{lin}(\psi_R) = 2\psi_3 + \psi_1 - 3$, $\psi'_D = \{\psi'_1\}$ with $1 \leq \psi'_1 \leq 5$ leads to $\text{lin}(\psi'_D) = \psi'_1 - 1$ thus $\mathcal{B}$ is $\psi'_1 - 1 = 4\lambda + 2\psi_3 + \psi_1 - 3$. The resulting assignment is shown in Table 4 and Figure 7. Because there are 5 targets and 4 available sources, the distribution cycles around the sources, and the first source processor set gets 2 targets.

Target		Source			Cycle
ψ'_1	$\text{lin}(\psi'_D)$	ψ_1	ψ_3	$\text{lin}(\psi_R)$	λ
1	0	1	1	0	0
2	1	2	1	1	0
3	2	1	2	2	0
4	3	2	2	3	0
5	4	1	1	0	1

Table 4: $\mathcal{B}$ target to source assignment for the running example

Proposition 3 (Target Affection) *Target processors are affected to* one *source processor among the replicated ones through $\mathcal{B}$:*

$$\forall \psi'_D, \exists! \psi_R \wedge \exists! \lambda | (\psi_R, \psi'_D, \lambda) \in \mathcal{B}$$

Proof: The linearization is dense. □

4 SPMD code generation

Let us now introduce the final polyhedron which integrates these choices and is used for the code generation:

Definition 4 (Elements $\mathcal{E}$) *With* $p = \psi \cup \psi' = \psi_D \cup \psi_R \cup \psi'_D \cup \psi'_R$*:*

$$\mathcal{E}(p, e, \lambda) = \mathcal{R}(p, e) \cup \mathcal{B}(\psi_R, \psi'_D, \lambda)$$

Polyhedron $\mathcal{E}$ is constrained enough so that there is only one possible sender (Proposition 5) for a given piece of data to be sent to all target processors requiring it. Thus a precise communication code can be generated: If (α, ψ, ψ') is a solution to $\mathcal{E}$, then ψ must send α to ψ'. Indeed, this polyhedron has the following properties:

Proposition 4 (Orthogonality of ψ'_R in $\mathcal{E}$)

$$\mathcal{E}(p, e, \lambda) = \mathcal{E}_{|\psi'_R}(p_D, \psi_R, e, \lambda) \times \mathcal{D}(\psi'_R)$$

Proof: Definition 4 and Proposition 1. □

Proposition 5 (One Sender) *For a required data on a target processor, there is only one sender defined in* $\mathcal{E}$*, which is independent of the replicated target* (ψ'_R)*:*

$$\forall(\psi'_D, \alpha) \in \mathcal{R}(p, e), \exists!\psi | (\psi, \psi'_D, \alpha) \in \mathcal{E}$$

Proof: Propositions 2, 3 and 4. □

Proposition 6 (Aggregation) *If a target processor requires two different pieces of data that can be sent by the same processor, then there is only one such processor:*

$$\forall\psi', \forall\alpha_i, \forall\alpha_j | (\exists\psi_D, (\psi', \psi_D, \alpha_i) \in \mathcal{E} \wedge (\psi', \psi_D, \alpha_j) \in \mathcal{E}) \Rightarrow (\exists!\psi | (\psi', \psi, \alpha_i) \in \mathcal{E} \wedge (\psi', \psi, \alpha_j) \in \mathcal{E})$$

Proof: Propositions 2 and 3: $\psi = \psi_D \cup \psi_R$, and ψ_R choice in $\mathcal{B}$ is independent of α. □

If all processors must enumerate all the integer solutions to polyhedron $\mathcal{E}$, this is equivalent to the runtime resolution technique and is very inefficient. Moreover, it would be interesting to pack at once the data to be sent between two processors, in order to have only one buffer for message aggregation. Therefore some manipulations are needed to generate efficient code.

Firstly, replicated dimensions of target processors (ψ'_R) are extracted from $\mathcal{E}$ as allowed by Proposition 4. This information is only required for broadcasting the message to the target processors. $\mathcal{E}_{|\psi'_R}$ stores the remaining information.

Secondly, in order to first enumerate the couples of processors that must communicate, and then to generate the associated message, a *super*set of these communicating processors is derived:

Definition 5 (Processors $\mathcal{P}$)

$$\mathcal{P}(\psi_D, \psi_R, \psi'_D, \lambda) = \mathcal{E}_{|\psi'_R, e}$$

This projection may not be exact[6]. $\mathcal{P}$ represents processors that *may* have to communicate: empty messages may be generated for processor couples in $\mathcal{P}$. To avoid sending and receiving these empty messages, while preserving the balance of messages, the following runtime technique is used: (1) in the send part, messages empty after packing are not sent; (2) in the receive part, messages are lazily received when some data must be unpacked. The technique is shown in Figure 11.

[6] A projection may be exact or approximate [2, 29], that is the integer solution to the projection may always reflects, or not, an integer solution to the original polyhedron.

```
– remapping of array A from processors Ps to processors Pt
– local declarations: As on source and At on target

if (I am in Ps) then – send part
    ψ = my id in Ps
    if (ψ ∈ P_{|ψ',λ}(ψ)) then – I may have to send something
        for (λ, ψ'_D) ∈ P_{|ψ'_R}[ψ] – enumerate target processors
            if (ψ'_R ≠ ∅ or pid(ψ) ≠ pid(ψ'_D)) then – some distributed targets
                empty = true
                for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ] – enumerate elements to send
                    pack As(local_source_address(e)) in buffer
                    empty = false – now the buffer is not empty
                endfor
                if (not empty) then broadcast buffer to ψ'_D × D(ψ'_R) except myself
            endif
        endfor
    endif
endif

if (I am in Pt) then – receive or copy part
    Allocate At
    ψ' = my id in Pt
    if (ψ'_D ∈ P_{|ψ'_R,ψ,λ}(ψ'_D)) then – I may have to receive something
        for (ψ, λ) ∈ P_{|ψ'_R}[ψ'_D] – enumerate source processors
            if (pid(ψ) ≠ pid(ψ')) then – non local, lazy reception and unpacking
                first = true
                for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ] – enumerate elements to receive
                    if (first) then receive buffer from ψ, first = false
                    unpack At(local_target_address(e)) from buffer
                endfor
            else – copy local data
                for e ∈ E_{|ψ'_R}[ψ, ψ'_D, λ]
                    At(local_target_address(e)) = As(local_source_address(e))
                endfor
            endif
        endfor
    endif
endif

if (I am in Ps) then Free As
```

Figure 11: SPMD remapping code

Thirdly, in a SPMD code executed in parallel, each (maybe virtual) processor plays a part (or none) in the processor grids `Ps` and `Pt`. Hence not all processors should enumerate the couples of communicating processors: processors in `Ps` [resp. `Pt`] are just interested in enumerating their matching target [resp. source] processors for sending [resp. receiving] data. $\mathcal{P}$ can be used to generate guards to select relevant processors and then to directly enumerate the sole matching processors in the other grid. At last, processors from different processor grids may be allocated to the same *physical* processor. Thus the code must not send a message from a processor to itself, but rather generate a local copy of the required elements instead.

Figure 11 shows the SPMD code generated with $\mathcal{P}$ and $\mathcal{E}$. The code is composed of a send and a receive part. The send part first selects the processors in `Ps`, and among them those which may communicate ($\mathcal{P}_{|\psi',\lambda}$). Then the corresponding target processors are enumerated (ψ'_D loop). If there is a broadcast or if the target and source *physical* processor differ, the data are packed in a message (e loop). Then the message is sent of not empty. Function local_source_address() computes the local address for a given array element on the source processors. If the local addressing scheme is integrated in the modelization, the local adress is directly enumerated in e.

The receive part is the dual of the send part. It selects the processors in `Pt`, and among them those processors which may have to receive some data. Then the corresponding senders are enumerated, and the messages are lazily received and unpacked to the local target array. If the sender was the processor itself, a local copy is performed: the communicating couple led to identical *physical* processors. The data is just copied from the source to target arrays. The copy is performed on the receive part in order not to delay the messages sending.

The generated code requires the enumeration of some polyhedra. Techniques based on Fourier elimination [21, 2] or a parametric simplex [13] generates code to *exactly* enumerate the solutions to a polyhedron. The correctness of the communications requires that the messages are packed and unpacked in the same order. This is enforced because the very same loop nest on $\mathcal{E}$ is generated for both packing and unpacking.

5 Optimality and discussion

For a given remapping, a minimal number of messages, containing only the required data, is sent over the network:

Theorem 2 (Only Required Data is Sent) *If the source and target processor grids are disjoined on the physical processors, only required data is communicated.*

Proof: $\mathcal{E}$ exactly describes the array elements and their mapping (derived from Proposition 2). Polyhedron scanning techniques *exactly* enumerate the elements in $\mathcal{E}$ and these elements *must* be communicated if the processors are disjoined. □

Theorem 3 (Minimum Number of Messages is Sent) *If the source and target processor grids are disjoined on the real processors, a minimal number of messages is sent over the network.*

Proof: Only required data is communicated (Theorem 2), all possible aggregations are performed (Proposition 6) and empty messages are not sent. □

Theorem 4 (Memory Requirements) *The maximum amount of memory required per* HPF *processor for a remapping is* $2 \cdot (memory(\texttt{As}) + memory(\texttt{At}))$.

Proof: Local arrays plus send and receive buffers may be allocated at the same time. The buffer sizes are bounded by the local array sizes because no more than owned is sent (even for broadcasts) and no more than needed is received (Theorem 2). □

- Special remappings that involve no communications can be automatically detected: if the target mapping is a particularization of the source mapping, *i.e.* all the needed data is locally available.
- The usual Fourier elimination technique needs to know the number of processors. However the parametric extension presented in [1] allows to generate code if this number is parametric.
- Since processor distributed dimensions are independent in $\mathcal{E}$, the practical complexity of the code generation for multiple dimensions roughly is the *number of dimensions* times the complexity of the code generation for one dimension. As expected, simple codes are generated for simple remappings, and more complicated ones for general cyclic distributions.

6 Experiments

The remapping generation technique is implemented in HPFC, our prototype HPF compiler. PVM is used for handling communications. HPFC aims primarily at demonstrating feasability and being portable, rather than achieving very high efficiency on a peculiar architecture. These new features were tested on a DEC Alpha farm at LIFL (Université de Lilles, France). The experimental results and derived data are presented in this section. They show some improvement over communications generated by the DEC HPF compiler, despite our high level implementation. Quite good performances for complex remappings compared to the simple and straightforward block distribution case were obtained. Experimental conditions and raw measures are presented and analyzed.

6.1 Experimental conditions

This section presents the hardware and software environment used for the tests, the measurements and the experiments.

DEC Alpha farm: 16 DEC 3000 model 400 AXP (512KB cache, 133MHz Alpha 21064) 64 MB memory workstations linked with a 100Mb/s/link[7] FDDI crossbar. Non dedicated machines.

Compilers: DEC Fortran OSF/1 `f77` version 3.5 with `"-fast -O3 -u"` options. DEC C OSF/1 `cc` with `"-O4"` option (for part of the hpfc runtime library). DEC HPF `f90` version FT1.2 with `"-fast -wsf` n`"` options for comparison with the remapping codes generated by HPFC, our prototype HPF compiler.

Communications: PVM version 3.3.9 standard installation, used with direct route option and raw data encoding. `PVMBUFSIZE` not changed. 1 MB intermediate buffer used to avoid packing each array element through PVM.

Transposition: A square matrix transposition was tested for various matrix sizes[8], processor arrangements and distributions. The time to complete `A=TRANSPOSE(B)` with `A` and `B` initially aligned was measured. It includes packing, sending, receiving and unpacking the data, plus performing the transposition. The code is shown in Figure 12. The 2D remapping *compilation* times ranged in $0.3 - 2.5$s on a SUN SS10. Other experiments with 1D remappings ranged in $0.2 - 1.5$s.

Measures: The figures present the best wall-clock execution time of at least 20 instances, after substraction of a measure overhead under-estimation. The starting time was taken

[7] 12.5 MB/s/link
[8] *i.e.* the number of lines and columns

size	bb	cc[a]	bc	c48c64	c5c7
256	0.0564	0.0623	0.0681	0.0681	0.1257
384	0.1218	0.1296	0.1238	0.1312	0.1471
512	0.2161	0.2175	0.2389	0.2405	0.3082
640	0.3287	0.3336	0.3912	0.4064	0.5792
768	0.7958	0.7783	0.5967	0.6108	0.8914
896	0.9168	0.9110	0.8183	0.8700	4.3824
1024	1.7519	1.8563	1.1189	1.1282	1.5577

Table 5: Transposition time (seconds) on P(2,2)

size	bb	cc	bc	c48c64	c5c7
256	0.0368	0.1803	0.0417	0.0524	0.0963
384	0.0875	0.4243	0.1120	0.0993	0.1256
512	0.1525	0.7834	0.1871	0.1871	0.2203
640	0.2423	1.2648	0.3072	0.3257	0.3706
768	0.3731	2.4780	0.5058	0.4692	0.5804
896	0.4838	-	0.6962	0.7001	3.3175
1024	0.6425	-	0.9480	0.8562	1.2486
1152	1.2037	-	1.1852	1.0270	1.3355
1280	1.3784	-	1.4008	1.3139	1.9587
1408	1.9197	-	1.6181	1.8723	2.1964
1536	2.9675	-	2.3970	2.2199	2.5921

Table 6: Transposition time (seconds) on P(2,3)

	P(2,3)		P(3,3)		P(3,4)		P(4,4)	
size	HPFC	DEC	HPFC	DEC	HPFC	DEC	HPFC	DEC
256	0.0368	0.0429	0.0372	0.0332	0.0257	0.0229	0.0411	0.0173
384	0.0875	0.0976	0.0698	0.0674	0.0433	0.0503	0.0593	0.0417
512	0.1525	0.1825	0.1215	0.1416	0.0892	0.0942	0.0909	0.0710
640	0.2423	0.3065	0.1947	0.2202	0.1272	0.1576	0.1007	0.1159
768	0.3731	0.4256	0.2757	0.2827	0.1868	0.2201	0.1495	0.1696
896	0.4838	0.5934	0.4383	0.4496	0.2512	0.3231	0.1911	0.2350
1024	0.6425	0.8575	0.4588	0.5970	0.3361	0.4279	0.2281	0.2911
1152	1.2037	1.0468	0.8086	0.7092	0.4128	0.5271	0.2900	0.3960
1280	1.3784	1.3036	0.9488	1.0020	0.5084	0.6749	0.3545	0.4913
1408	1.9197	1.5641	1.0522	1.1953	0.6295	0.8320	0.4312	0.5966
1536	2.9675	1.9561	1.8209	1.3236	1.0120	0.9819	0.8147	0.7187
1792	-	-	2.1088	2.0908	1.6474	1.4240	1.0011	1.0085
2048	-	-	3.3807	2.8654	2.1184	1.8043	1.8235	1.3663

Table 7: Transposition time (seconds) for (block,block) distributions

[a] Invariant code motion and some code transformations where performed by hand for this distribution

```
      real*8 A(n,n), B(n,n)
chpf$ dynamic B
chpf$ template T(n,n)
chpf$ processors P(...)
chpf$ distribute T(...) onto P
chpf$ align A, B with T
      ...
c
c A = TRANSPOSE(B)
c
c first align A and B transpose
c
chpf$ realign B(i,j) with T(j,i)
c
c now the assignment, everything is local
c
chpf$ independent(j, i)
      do j=1, n
        do i=1, n
          A(i,j) = B(j,i)
        enddo
      enddo
c
c DONE
c
      ...
```

Figure 12: Remapping-based transpose code for HPFC

between two global synchronizations. The final time was taken after an additional global synchronization.

Raw measures for transpositions are displayed. Tables 5 and 6 show the transposition times for various matrix sizes and distributions. The column heads describe the distribution of the array dimensions: for instance `c5c7` stands for `(cyclic(5),cyclic(7))`. Table 7 show the `(block,block)` transposition time for various array arrangements, involving up to 16 processors. These raw measures are analyzed in the next section.

6.2 Performance analysis

From the previous raw measures, derived data are presented in Figures 13, 14 and 15. For comparison purposes, we introduce the transposition speed per processor, expressed in MB/s/pe[9]. This unit is independent of the matrix size n, the type length[10] l and the number of processors p involved. If t is the measured time, then speed s is defined as:

$$s = \frac{l.n^2}{2^{20}.p.t}$$

Comparable performances are obtained for different matrix sizes, processor arrangements and distributions. Simple transpositions on `(block,block)` distributed arrays perform generally better than others. However the *actual* amount of data that is communicated vary from one distribution to another, what is not taken into account.

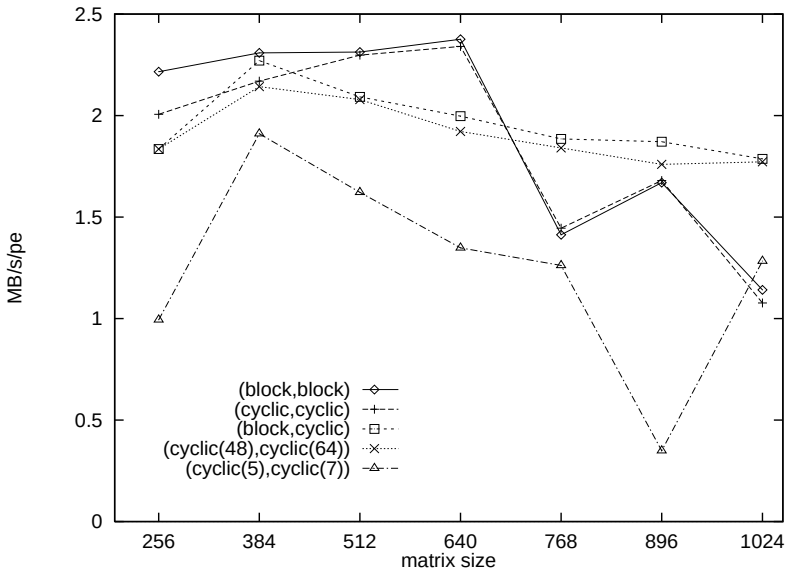

Figure 13: Transposition speed per processor on `P(2,2)`

Figure 13 displays the performances on `P(2,2)`. Cases `bb` and `cc` are very similar: Indeed, in both cases 2 processors must exchange all their local data and 2 do not have to communicate at all, thus the generated codes are very similar. Also two degradations due to PVM are noticeable for large matrix sizes, when the amount of communication steps over 1 MB and 2 MB. `bc` and `c48c64` show similar performances. `c5c7` is a tricky case, and the enumeration costs are higher.

[9] MB stands for Mega Bytes, and $1M = 2^{20} = 1\,048\,576$

[10] $l = 8$ in our experiments based on `real*8` arrays.

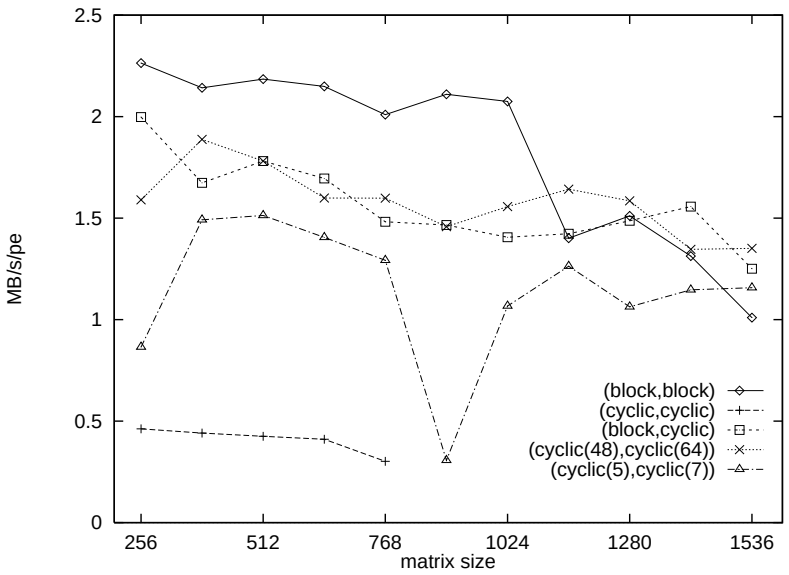

Figure 14: Transposition speed per processor on `P(2,3)`

Figure 14 presents quite similar results on `P(2,3)`, but for the `cc` case. Additional transformations [3], which are not yet implemented in the prototype, are needed to extract the lattice of accessed elements.

Finally Figure 15 shows performance of `(block,block)` transpositions on various processor arrangements for HPFC and the DEC HPF compiler. The `transpose` intrinsic is serialized according to the release notes, so it was not used. Since the `independent`, `realign` and `redistribute` directives are not implemented, only the available HPF `forall` instruction was used to transpose the matrix. Our performance is degraded for large matrix sizes because of the PVM 1 MB buffer size limit. Also the more processor the larger the matrix size is needed to get comparable speeds. Transposition speed based on our code show 20-30% improvements over the DEC compiler, up to PVM buffer size problems. However these results are not comparable: our code is a higher level one, based on PVM, and simple standard optimizations would be usefull for the compiling enumeration code efficiently.[11]

Complex remappings show quite good results with respect to simple ones. However the performances should be compared somehow to the peak 12.5 MB/s/link available. Some measurements show that the PVM overhead represents up to 80% of the measured time. A more aggressive buffer management for the remappings and the `PvmDataInPlace` option [4] may reduce this overhead. Generating code closer to the machine would also help, but at the price of portability. The experiments also show that lattice detection is an important issue for generating good code, and that optimizations such as invariant code motion can have a great influence on the performances of the polyhedron enumeration code.

Conclusion

A general compilation technique was presented to handle HPF remappings efficiently. This technique is implemented in HPFC, our prototype HPF compiler. Portable PVM-based [16] code is generated. The remapping code generation problem is put in a single linear framework that deals with all HPF issues such as alignments or general cyclic distributions. Optimality results

[11] This point will be detailed in the final version of the paper.

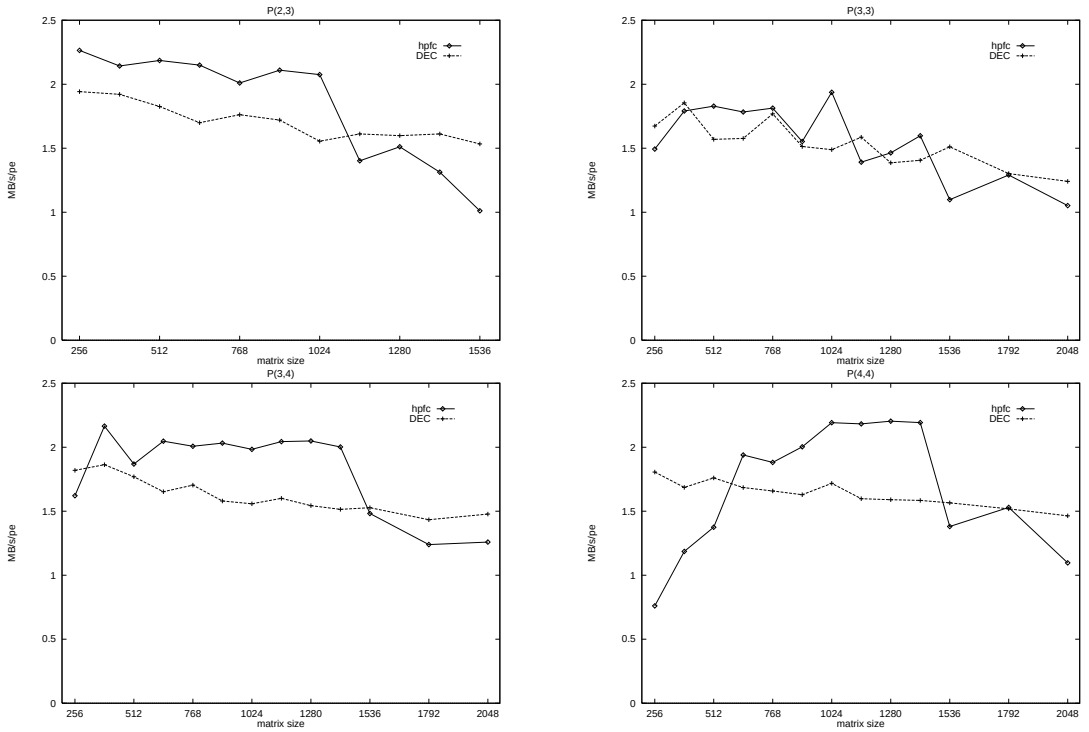

Figure 15: Transposition speed per processor for (block,block) distributions

were presented. Namely, a minimal number of messages, containing only the required data, is sent over the network. Thus the technique minimizes both the effects of latency and bandwith-related of the network, through message aggregation and exact enumeration of elements to be sent. Moreover load balancing issues are discussed and (possibly partial) broadcasts are used when possible.

However, there is still a need for runtime support. Some HPF programs may instantiate too many mapping parameters at runtime, making the parametric compile time code generation phase too tricky.

Future work includes:

- a new buffer management to use the PVM *in place* option.
- lattice extraction to generate better polyhedron enumeration codes.
- mappings code motion to reduce the number of executed remappings,
- reducing the remapped element set to what is used through advanced compile time analyses [12, 11].
- generating temporary copies for read-only remapped arrays to avoid backward remappings,
- compiling for other models, such as get/put/synchro communications.

Acknowledgements

We are thankful to (in alphabetical order) François BODIN for informal discussions, Béatrice CREUSILLET for pointers, Jean-Luc DEKEYSER for access of the Alpha farm, François IRIGOIN

for the improvements he suggested, Pierre JOUVELOT for corrections, Philippe MARQUET for technical support on the farm, William PUGH for suggestions and Xavier REDON for comments.

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. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales.* PhD thesis, Université Paris VI, March 1991.

[3] 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. Also available as TR EMP A/250/CRI on `http://www.cri.ensmp.fr`.

[4] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. Recent Enhancements to PVM. *Int. J. of Supercomputer Applications and High Performance Computing*, 9(2):108–127, summer 1995.

[5] 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.

[6] 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. In *Symposium on Principles and Practice of Parallel Programming*, 1993.

[7] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Array distribution in data-parallel programs. In *Language and Compilers for Parallel Computing*, pages 6.1–6.17, August 1994.

[8] Fabien Coelho. Étude de la Compilation du High Performance Fortran. Master's thesis, Université Paris VI, September 1993. Rapport de DEA Systèmes Informatiques. TR EMP E/178/CRI.

[9] 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.

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

[11] Béatrice Creusillet. IN and OUT array region analyses. In *Workshop on Compilers for Parallel Computers*, June 1995.

[12] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Language and Compilers for Parallel Computing*, August 1995.

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

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, May 1993. *Version 1.0.*

[15] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivation Examples.* Rice University, Houston, Texas, November 1994.

[16] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[17] 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.

[18] S.K.S. Gupta, C.-H. Huang, and P. Sadayappan. Implementing Fast Fourier Transforms on Distributed-Memory Multiprocessors using Data Redistributions. *Parallel Processing Letters*, 4(4):477–488, December 1994.

[19] S.K.S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. TR 19, Department of Computer and Information Science, The Ohio State University, 1994.

[20] Semma Hirannandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Advanced Compilation Techniques for Fortran D. CRPC-TR 93338, Center for Research on Parallel Computation, Rice University, October 1993.

[21] 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.

[22] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, June 1991.

[23] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Inc., Engelwood Cliffs, 1979.

[24] Ken Kennedy and Ulrich Kremer. Automatic Data Layout for High Performance Fortran. CRPC-TR94 498-S, Center for Research on Parallel Computation, Rice University, December 1994.

[25] 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 Vol. 30:8.

[26] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Language*, pages 194–206, 1973.

[27] Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele, and Mary Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[28] 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*, June 1991.

[29] William Pugh. A pratical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, August 1992.

[30] Shankar Ramaswamy and Prithviraj Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[31] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation and evaluation. In *Language and Compilers for Parallel Computing*, August 1993.

[32] Ernesto Su, Antonio Lain, Shankar Ramaswamy, Daniel J. Palermo, Eugene W. Hodges IV, and Prithviraj Banerjee. Advanced Compilation Techniques in the Paradigme Compiler for Distributed-Memory Multicomputers. In *ACM International Conference on Supercomputing*, pages 424–433, July 95.

[33] Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. Runtime array redistribution in HPF programs. In *Scalable High Performance Computing Conference*, pages 309–316, 1994.

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

[35] Vincent Van Dongen. Array redistribution by scanning polyhedra. In *PARCO*, September 1995.

[36] 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.

[37] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran - A Language Specification. ftp `cs.rice.edu public/HPFF/papers/vf.tex`, 1992. Version 1.1.