Optimal Compilation of HPF Remappings (Extended Abstract)

—

Report A–274–CRI

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`, FTP: `anonymous@ftp.cri.ensmp.fr`

July 18, 1995

Abstract

Applications with varying array access patterns in a parallel distributed memory machine require to dynamically change array mappings. HPF (High Performance Fortran) provides such remappings, possibly on partially replicated data, 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 reason. 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 naturally appear in procedures. The second phase generates the actual remapping code and takes advantage of replications to shorten the remapping time. A minimal number of messages, containing only the required data, is sent over the network.

Keywords:

HPF Compilation, Array remappings, Distributed Memory MIMD machines.

Number of words:

4984 words, not including figures, references, and LaTeX directives.

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

Introduction

Many applications require the ability to have different array mappings at different phases of computations for efficient execution on distributed-memory parallel machines, such as the CRAY T3D or the IBM SP2. ADI (Alternating Direction Integration) and FFT (Fast Fourier Transform) are examples of such applications [16]. Data replication is also useful to share data between processors. The combination of both features enables elegant programming techniques: for instance, the distributed definition of an array and its efficient use after replication. A parallel matrix multiplication accesses a whole row and column of data to compute each single element of the matrix, hence the need to remap data with some replication. Moreover, automatic tools for data layout [22, 6] may suggest data remappings between computation phases of an application. Thus handling data remappings is an important issue for high performance computing.

HPF (High Performance Fortran [13, 25]) is a data-parallel language based on Fortran 90. It targets distributed memory parallel architectures. It provides standard directives to declare processor sets and array mappings onto these. HPF mappings may involve some replication. The user may dynamically change array mappings in HPF: explicitly with the `realign` and `redistribute` *executable* directives; implicitly at procedure calls and returns, when prescriptive mappings are declared for arguments.

However useful these features are, they are perceived as difficult to compile efficiently and thus are left out of the HPF subset or the currently discussed HPF kernel [14]. If not supported, or even not well supported, remappings will not be used for applications aiming at high efficiency, indeed all HPF applications. It also decreases the number of codes that can be efficiently ported to HPF. The key issues to be dealt with are the reduction of the runtime overhead induced by remappings, and the treatment of the rich variety of HPF mappings.

Any technique that claims to handle all HPF array assignments may be used to compile remappings, since communications involved in a remapping are equivalent to an array assignment $A = B$, where B (resp. A) has the source (resp. target) mapping. Such techniques are based on finite state machines [5, 18, 23], closed form approaches [15, 29, 17], diophantine equations [26, 4, 34] or polyhedra [1, 3, 32, 30]. However none of these techniques does take advantage of replications and broadcasts. Also issues such as handling different processor sets, multidimensional distributions, communication set generation and allocation of temporary arrays are not all clearly and efficiently addressed in these papers, therefore the need for dedicated techniques, that can include more optimizations.

In [31] a runtime library support is suggested. Algorithms are presented for simple cases involving no shape changing (the distributed dimensions are the same for both source and target mappings), no alignment and no replication. Multidimensional remappings are decomposed into 1-D remappings, hence resulting in several remappings at runtime. In the PARADIGME project [28], a compile time technique based on *ad hoc* data descriptors called PITFALLS is presented. Alignment, replication and shape changing are not considered. For the EPPP project [33], a polyhedron based approach is outlined. Only realignments with a fixed block-cyclic distribution onto a 1-D processor array are handled. However the alignments, unlike HPF, involve arbitrary affine functions.

Remapping overheads are attacked at different levels by the compilation technique presented in this paper. First, at a global level, all useless remappings are removed. Such remappings arise naturally in programs: for instance, from a template redistribution if not all aligned arrays are used afterward. This optimization is presented in Section 1. Second, in Section 2, the code generation problem for a given useful remapping is fitted into a single powerful linear framework, which integrates all issues (alignment strides, general cyclic distributions, partial replication, different processor sets and local addressing). Load balance and use of broadcasts are discussed in Section 2.4 for an SPMD code generation targeting a message-passing distributed-memory MIMD architecture with non-blocking sends and blocking receives. Correctness and optimality results are presented and discussed for both phases of the compilation.

This technique has been successfully implemented in HPFC [7, 8, 9], a prototype HPF compiler developed within the PIPS project [20]. To our knowledge, it is the first to perform global optimizations to remove useless remappings, to integrate all HPF mapping issues in a single framework, to address load balancing and possible broadcasts in the generated code and to present optimility results.

1 Remapping Graph

Useless remappings may appear naturally in HPF programs. First, the change of the alignment and the distribution of an array necessitates a `realign` and a `redistribute`, hence resulting in two remappings if no special care is taken. Second, the redistribution of a template [1] 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. These useless remappings are not taken from badly written codes. 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

[1]Even when no templates are used [35] array alignment generates the problem

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 graph, and optimizations performed on this graph are presented. This approach only deals with descriptive and prescriptive mappings, *i.e.* when the compiler is aware of data distributions.

1.1 Definition

Let us introduce the remapping graph $\mathcal{G}_R$. This graph is a subgraph of the control flow graph, usually much smaller. The vertices of the graph are the remapping statements, *i.e.* `realign`s and `redistribute`s. An edge denotes a possible path in the control 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 $\mathcal{G}_R$ is associated:

- the set of remapped arrays $\mathrm{S}(v)$
- for all remapped arrays `A` in this set:
  - one leaving mapping[2] for the statement: $\mathrm{L}_{\mathtt{A}}(v)$
  - whether it may be referenced after the remapping: $\mathrm{Used}_{\mathtt{A}}(v)$
  - the set of mappings for `A` that may reach v: $\mathrm{R}_{\mathtt{A}}(v)$

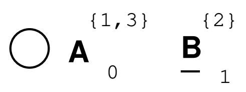

Figure 2: A simple vertex

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 mapping,leaving mapping).

1.2 Construction

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

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

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`.

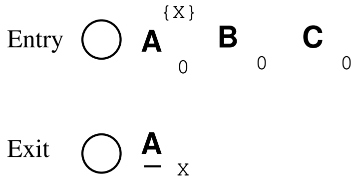

Figure 3: $\mathcal{G}_R$ for `remaps` at construction start

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 $(v, \mathtt{A})$, 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 corresponding w vertex in $\mathcal{G}_R$ if necessary. Add `A` to $\mathrm{S}(w)$ and compute $\mathrm{L}_{\mathtt{A}}(w)$ if necessary, and $(w, \mathtt{A})$ is a new couple to be explored later on. Add $\mathrm{L}_{\mathtt{A}}(v)$ to $\mathrm{R}_{\mathtt{A}}(w)$. The resulting remapping graph for `remaps` is shown in Figure 4.

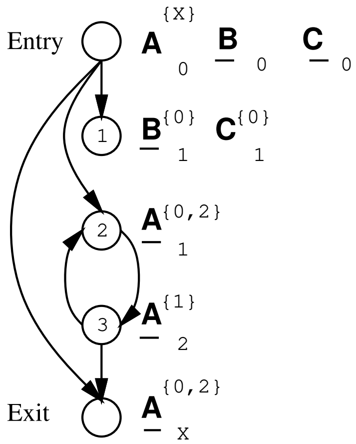

Figure 4: Initial $\mathcal{G}_R$ for `remaps`

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.

1.3 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, simply by deleting the leaving mapping for those vertices and arrays.

  $$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \text{not } \mathrm{Used}_{\mathtt{A}}(v) \Rightarrow \mathrm{L}_{\mathtt{A}}(v) = \emptyset$$

  As a convention in the interpretation of the remapping graph, remappings at vertex v for array $\mathtt{A} \in \mathrm{S}(v)$ will not be generated if $\mathrm{L}_{\mathtt{A}}(v) = \emptyset$.

- Second, recompute the mappings that may reach each vertex. This is a forward may data flow problem [24, 21] on $\mathcal{G}_R$:

  - initialization: Used 1-step reaching mappings

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

  - 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.

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:

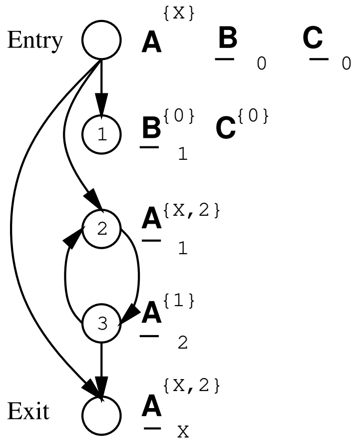

Figure 5: Optimized $\mathcal{G}_R$ for `remaps`

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):* $\forall v$, $\forall \mathtt{A} \in S(v) \wedge Used_{\mathtt{A}}(v)$, $\forall a \in R_{\mathtt{A}}(v)$,
$\exists w$ *and a path from* w *to* v *in* $\mathcal{G}_R$, *so that* $a \in L_{\mathtt{A}}(w)$ *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.

1.4 Discussion

If subroutine local arrays are not used at the beginning of the subroutine 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 instantiate the mapping and to compute the needed remapping routines.

The remapping graph was presented at an intraprocedural level. It would be natural to extend it to the interprocedural level, by providing a summary of the entry and exit remappings to be used at the call sites for optimizations. 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, and $\mathcal{G}_R$ would be simplified.

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 propagation phase of the optimization. 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.

Finally, 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 Code generation for one remapping

This section presents the compilation technique second phase for generating efficient code to remap data onto distributed parallel machines. The technique deals with arbitrary remappings involving partial replications, alignments, general cyclic distributions onto processor sets with different shapes.

```
      array A(n)
chpf$ template T(n,n,n)
chpf$ align A(i) with T(*,i,*)
chpf$ processors P(2,2,2)
chpf$ distribute T(block, block, block) onto P
chpf$ template T'(n,n)
chpf$ processors P'(5,2)
chpf$ distribute T'(cyclic, block) onto P'
      ...
c
c here is the realignment we are interested in:
c
c     P(*,block,*) -> P'(cyclic,*)
c
chpf$ realign A(i) with T'(i,*)
      ...
```

Figure 6: Running example for code generation

Let us consider the example in Figure 6. Vector `A` is realigned from 3-D processor grid `P` to 2-D processor grid `P'` through templates `T` and `T'`. `A` is replicated on dimensions 1 and 3 of `P`, and 2 of `P'`: for each x, processors `P(*,x,*)` share the same data (idem for `P'(x,*)`). The initial block distribution is changed to cyclic one. This example is deliberately contrived, and designed to show all the capabilities of our algorithm. Remappings for realistic applications are not likely to present all these difficulties at once, but they should frequently include one of them.

Linear algebra provides a powerful framework to characterize array element sets, to represent HPF directives, to generate efficient code, to define and introduce optimizations and to make possible proof of the compilation scheme. Our compilation scheme uses polyhedra to represent the HPF data remapping problem.

Firstly, the linear formulation of the redistribution problem is presented and illustrated on the running example. Secondly, optimizations taking into account (1) particular distribution of data onto the processors such as replication and (2) efficient communication capabilities of the distributed memory machine such as broadcast are added to the the compilation scheme. Thirdly, the algorithm generating an SPMD code is presented as well as optimality properties. Finally, some extensions are presented.

2.1 Notations

The notations that are used in this section are introduced hereafter:

- All manipulated variables in this paper are integer variables.
- $\mathcal{D}(V)$ denotes the declaration constraints on variables V, *i.e.* lower and upper bounds of cartesian constraints: $\mathcal{D}(V) = \cup_{v \in V}\{\min(v) \leq v \leq \max(v)\}$.
- $|v|$ represents the extent of Variable v, *i.e.* $|v| = \max(v) - \min(v) + 1$.
- Primes [resp. non-primes] will denote variables related to the target [resp. source] mapping, when relevant.
- α, $\theta[']$, $\psi[']$ represent sets of variables for array, template and processor dimensions. Subscripted versions denote individual variables for a given dimension.
- $\gamma[']$, $\delta[']$ are cycle numbers and block offsets introduced to model HPF distributions.
- $\psi_D[']$ [resp. $\psi_R[']$] define sets of variables for dimensions of processor grids on which distribution [resp. replication] occur. $\psi['] = \psi_D['] + \psi_R[']$

Let $\mathcal{X}(V)$ be a system of linear equalities and inequalities on variables in V. These constraints implicitly defines a polyhedron on variables in V, *i.e.* the set of integer vectors that are solutions to $\mathcal{X}$.

- $Z \in \mathcal{X}[W](V)$ is the enumeration of the solution of $\mathcal{X}$ with the $W \subset V$ variables considered as parameters ($V = W + Z$). That is a parametric polyhedron on variables Z.
- $\mathcal{X}_{|Y}(V)$ is the system after the projection of variables $Y \subset V$. A projection may be exact or approximate [2, 27], that is the integer solution to the projection may always reflects, or not, an integer solutions in the original polyhedron.
- $\mathcal{X}_3(Z_1 \cup Z_2) = \mathcal{X}_1(Z_1) \cup \mathcal{X}_2(Z_2)$ denotes the union of systems $\mathcal{X}_1$ and $\mathcal{X}_2$. This correspond to the intersection of the corresponding polyhedrons.
- Cartesian product $\times$ denotes the disjoined union, *i.e.* in the preceding item if $\mathcal{X}_1$ and $\mathcal{X}_2$ define constraints on disjoined set of variables (if $Z_1 \cap Z_2 = \emptyset$).

Definition 1 (linearization) *Let V be a list of bounded variables $(v_1, v_2, \ldots, v_n)$, constrained in a cartesian domain. The linearization function lin is recursively defined as:*

- $\mathit{lin}(\emptyset) = 0$
- $\mathit{lin}(V) = |v_1| \cdot \mathit{lin}\,(V - \{v_1\}) + v_1 - \min(v_1)$

The cardinal $||$ is conventionnaly extended to linearized sets:

- $|lin(\emptyset)| = 1$
- $|lin(V)| = \prod_{v \in V} |v|$

It is used to compute the address of array elements indexed by variables in V. However it will be used to linearize subset of variables in order to break the dimension structuration.

As an example, linearizing list $V = (\psi_1, \psi_3)$ where $1 \leq \psi_1 \leq 2, 1 \leq \psi_3 \leq 2$ gives the linearization equality: $\mathrm{lin}(V) = 2\psi_3 + \psi_1 - 3$.

2.2 Linear formalization

HPF directives, program variable declarations and the local addressing scheme are translated into linear constraints, as suggested in [3, 9]. This gives a linear description of the data distribution and of the communication problem, that is the enumeration of the elements to be sent and received. Figure 7 shows the linear constraints derived for the running example presented in Figure 6. These equations are detailed in the appendix.

$$
\begin{array}{rl}
\mathtt{A}(\alpha_1) & 1 \leq \alpha_1 \leq 100 \\
\mathtt{T}(\theta_1,\theta_2,\theta_3) & 1 \leq \theta_1 \leq 100, 1 \leq \theta_2 \leq 100, 1 \leq \theta_3 \leq 100 \\
\mathtt{T'}(\theta'_1,\theta'_2) & 1 \leq \theta'_1 \leq 100, 1 \leq \theta'_2 \leq 100 \\
\mathtt{P}(\psi_1,\psi_2,\psi_3) & 1 \leq \psi_1 \leq 2, 1 \leq \psi_2 \leq 2, 1 \leq \psi_3 \leq 2 \\
\mathtt{P'}(\psi'_1,\psi'_2) & 1 \leq \psi'_1 \leq 5, \;\; 1 \leq \psi'_2 \leq 2 \\
\texttt{align A(i) with T(*,i,*)} & \theta_2 = \alpha_1 \\
\texttt{align A(i) with T'(i,*)} & \theta'_1 = \alpha_1 \\
\begin{array}{c}\texttt{distribution of A} \\ \texttt{via T onto P}\end{array} & \left\{ \begin{array}{l} \theta_2 = 50\psi_2 + \delta_2 - 49, \\ 0 \leq \delta_2 < 50 \end{array} \right. \\
\begin{array}{c}\texttt{distribution of A} \\ \texttt{via T' onto P'}\end{array} & \{\; \theta'_1 = 5\gamma'_1 + \psi'_1 \\
\begin{array}{c}\texttt{Replicated and distributed} \\ \texttt{source processor dimensions}\end{array} & \psi_R = \{\psi_1, \psi_3\}, \psi_D = \psi_2 \\
\begin{array}{c}\texttt{Replicated and distributed} \\ \texttt{target processor dimensions}\end{array} & \psi'_R = \psi'_2, \psi'_D = \psi'_1 \\
\begin{array}{c}\texttt{communication cyclic} \\ \texttt{distribution equation}\end{array} & \left\{ \begin{array}{l} \psi'_1 = 2 \cdot \psi_3 + \psi_1 + 4 \cdot \lambda - 2 \\ 1 \leq \psi'_1 \leq 5, 1 \leq \psi_1 \leq 2, 1 \leq \psi_3 \leq 2 \\ 0 \leq \lambda < 4 \end{array} \right.
\end{array}
$$

Figure 7: Linear constraints induces by the running example

These linear constraints define a polyhedron on α, θ, ψ, γ and δ and corresponding primed variables, that include all HPF constraints defining a remapping. However some of the variables are of no interest for the code generation, and can be exactly projected. They are the θ's, δ's on cyclic-1 distributions and γ's on block distributions. Let p be the set of variables on the processors ($p = \psi + \psi'$) and $e = \{\alpha, \ldots\}$ the other variables. Let $\mathcal{R}(p, e)$ be the final polyhedron. $\mathcal{R}$ models all HPF constraints for the remapping. Solutions to this polyhedron reflect the array elements (α) and their presence on the source (ψ) and target (ψ') processor sets. $\mathcal{R}$ satisfies some properties because of the HPF semantics.

Proposition 1 (replication independence) *Processor variables on replicated dimensions are disjoined from others in $\mathcal{R}$: $\mathcal{R}(e,p) = \mathcal{R}_{|p_R}(e,p_D) \times \mathcal{D}(p_R)$*

Proof: HPF constraints, structure of the system.□

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

$$\forall \alpha \in \mathcal{D}(A), \exists!(e,p_D) | (e,p_D) \in \mathcal{R}_{|p_R}(e,p_D)$$

I.e. *apart from replicated dimensions, only one processor owns a data on the source and target processor grids.*

Proof: HPF constraints.□

2.3 Load balancing and Use of broadcasts

The SPMD code can be generated by enumerating the solutions to the polyhedron, that associates array elements (α) to possible senders on the source processor grid P (ψ) and the possible receivers on the target processor grid P′ (ψ'). However $\mathcal{R}$ is not constrained enough for that purpose: any replicated (ψ_R) processor is a candidate to send a given piece of data (α). One must be chosen. The naive way to do so is to arbitrarily fix ψ_R with some valid value. However all but one source processor on the replicated dimensions would be unemployed. There is an opportunity for load balancing: each source processor owning the same data may generate messages for different target processors requiring these data. The following constraint achieves this goal:

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

The formula describes a distribution assigning distributed target processors to replicated source processors in a cyclic fashion. The rational 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 ($\mathrm{lin}(\psi_R)$), as well target processors requiring different elements ($\mathrm{lin}(\psi'_D)$) in order to generate the cyclic distribution.

The processors in Figures 8 and 9 identically shaded represent the set of source and target processors that contain the same replicated array elements.

In order to minimize the number of generated messages, replicated dimensions on target processors ψ'_R do not take place in the equation. Instead of individual messages to ψ'_R processors, broadcast messages are introduced in the generated code. In our example, each source processor should send the same pattern of array elements to the target processors on the same row. Therefore to reduce the number of generated messages, Processor (1,1,1) broadcasts the communication set to the target Processors (1,*) and (5,*), Processor (1,1,2) broadcasts the communication set to the target Processors (2,*)…as depicted in Figure 10.

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.□

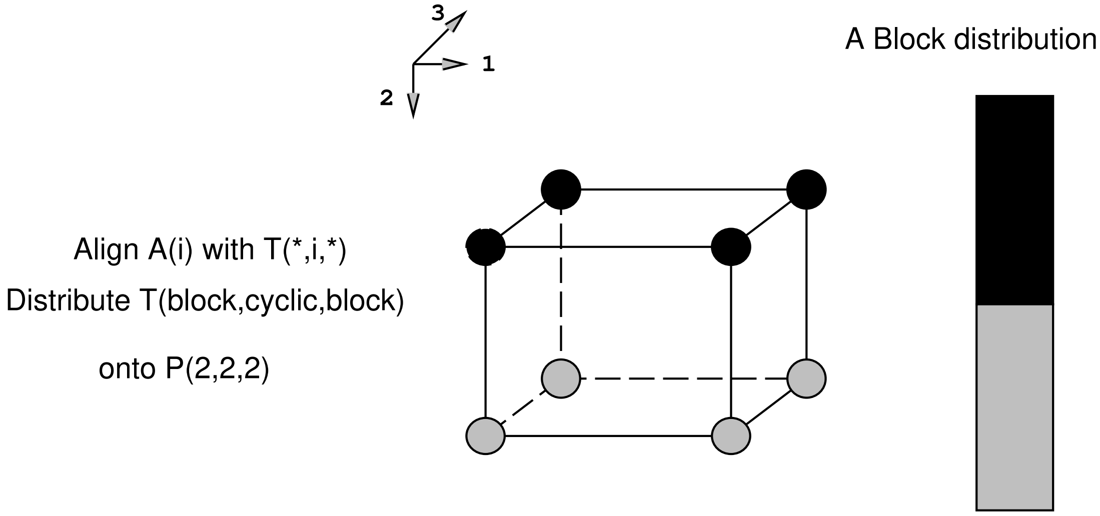

Figure 8: Distribution onto source processors

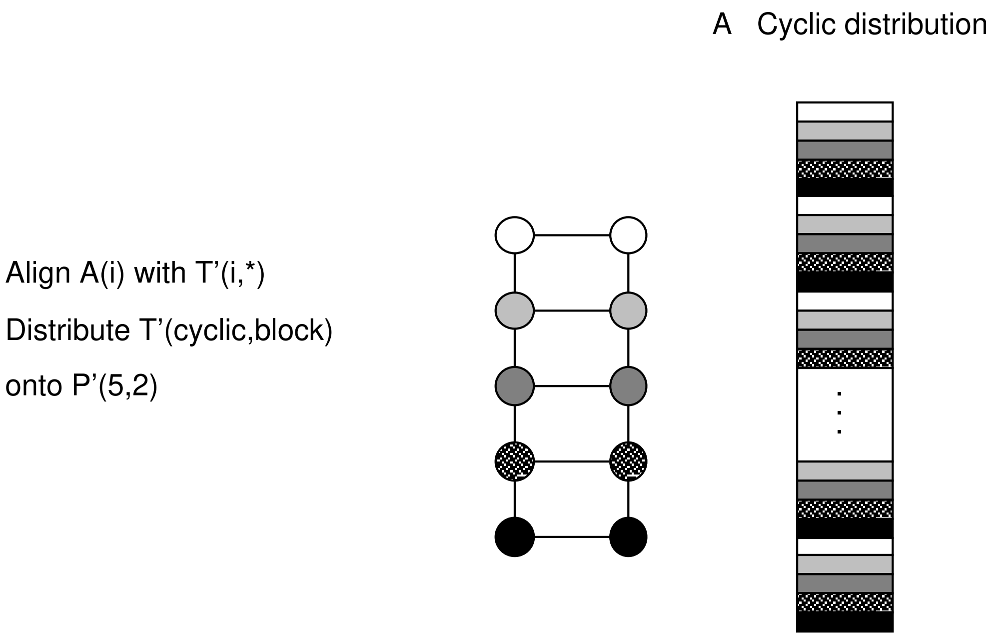

Figure 9: Distribution onto target processors

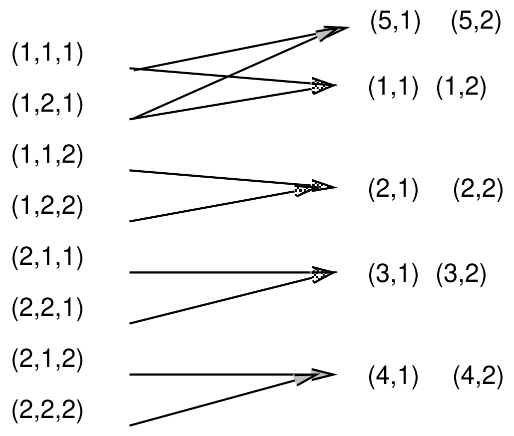

Figure 10: Communications and target processors

2.4 SPMD code

Let us consider polyhedron $\mathcal{E}$:

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

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

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

Proof: Propositions 2, 3 and 4.□

Proposition 6 (aggregation) *If a target processor requires two different piece of data that may be sent by the same processor, then they are sent by the same source processor:*
$\forall \psi', \forall \alpha_i, \forall \alpha_j | (\exists \psi_D, (\alpha_i, \psi', \psi_D) \in \mathcal{E} \wedge (\alpha_j, \psi', \psi_D) \in \mathcal{E})$
$\Rightarrow (\exists! \psi_R | (\alpha_i, \psi', \psi) \in \mathcal{E} \wedge (\alpha_j, \psi', \psi) \in \mathcal{E})$

Proof: Propositions 2 and 3: ψ_R choice in $\mathcal{B}$ is independent of array element α.□

If all processors must enumerate all the integer solutions to the polyhedron, 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 optimizations are needed to generate efficient code from $\mathcal{E}$.

Firstly, replicated dimensions of target processors (ψ'_R) are extracted from $\mathcal{E}$(Proposition 4). This information is only required for broadcasting, *i.e.* once message must be sent to all waiting target processors. The remaining information is in $\mathcal{E}_{|\psi'_R}$. Secondly, in order to enumerate first the couples of processors that must communicate, and then to generate the associated message, we can compute a view $\mathcal{P}$ of these communicating processors:

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

This projection may not be exact. $\mathcal{P}$ represents a set of 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 techniques are 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.

Thirdly, in an SPMD code executed in parallel, each (maybe virtual) processor plays a part (or none) in the processor grids `P` and `P'`. Hence not all processors should enumerate the couples of communicating processors: processors in `P` [resp. `P'`] 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 enumerate directly only the 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 avoid sending a message from a processor to itself, and 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 part and a receive part. The send part first selects the processors in `P`, and among them those which may communicate ($\mathcal{P}_{|\psi',\lambda}$). Then the corresponding target processors are enumerated (ψ'_D loop). The data for each of these processors are packed (e loop), and sent to all the processors waiting for them, if the generated message is not empty. local_source_address() computes the local address for a given array local element on the source processors.

The receive part is the dual of the send part. It selects the processors in `P'`, and among them those 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 leads to identical *real* processors. The data are just copied from the source to target array. The copy is performed on the receive part in order not to delay the messages sending.

The generated code necessitates the enumeration of some polyhedra. Techniques for generating such codes are based on Fourier elimination [19, 2] or a parametric simplex [12]. Such techniques generate code to enumerate *exactly* the solutions to a polyhedron. The correctness of the communications requires that the messages are packed and unpacked in the same order. This is insured because the same loop on $\mathcal{E}$ is generated for both packing and unpacking.

2.5 Results and Extensions

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

```
– remapping of array A from processors P to processors P'
– local declarations: A_s on source and A_t on target
if (I am in P) then
    – send part
    $\psi$ = my id in P
    if ($\psi \in \mathcal{P}_{|\psi',\lambda}(\psi)$) then – I may have to send something
        for ($(\lambda, \psi'_D) \in \mathcal{P}_{|\psi'_R}[\psi]$) – enumerate target processors
            empty = true
            for $e \in \mathcal{E}_{|\psi'_R}[\psi, \psi'_D, \lambda]$
                pack A_s(local_source_address(e)) in buffer
                empty = false – the buffer is not empty
            endfor
            if (not empty) then broadcast buffer to $\psi'_D \times \mathcal{P}'_R(\psi'_R)$ but myself
        endfor
    endif
endif
if (I am in P') then
    – receive or copy part
    Allocate A_t
    $\psi'$ = my id in P'
    if ( $\psi'_D \in \mathcal{P}_{|\psi'_R,\psi,\lambda}(\psi'_D)$ ) then – I may have to receive something
        for $(\psi, \lambda) \in \mathcal{P}_{|\psi'_R}[\psi'_D]$ – enumerate source processors
            if (pid($\psi$) $\neq$ pid($\psi'$)) then – non local, lazy reception and unpacking
                first = true
                for $e \in \mathcal{E}_{|\psi'_R}[\psi, \psi'_D, \lambda]$
                    if (first) then receive buffer from $\psi$, first = false
                    unpack A_t(local_target_address(e)) from buffer
                endfor
            else – local, copy
                for $e \in \mathcal{E}_{|\psi'_R}[\psi, \psi'_D, \lambda]$
                    A_t(local_target_address(e)) = A_s(local_source_address(e))
                endfor
            endif
        endfor
    endif
endif
if (I am in P) then Free A_s
```

Figure 11: SPMD remapping code

Theorem 2 (only required data 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 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.□

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 are locally available.

In the send part of the generated code, the packing overhead of array elements A_s induced when there is no target processor excepted the source processor, can be eliminated. This case only occurs when there is no replicated target processors. Thus, a simple guard $\mathrm{pid}(\psi) \neq \mathrm{pid}(\psi'_D)$ can be added outside of the packing loop to achieve this goal.

The usual Fourier elimination technique needs to instantiate the number of processors. However the parametric extension presented in [1] allows to generate code if this number is parametric.

Since processor dimensions are independent in $\mathcal{E}$, the practical complexity of the code generation for multiple dimensions is the *number of dimensions* times the complexity of the code generation for one dimension.

Conclusion

Experimental results will be presented in the final version of this paper. Experiments will show compilation and execution times for various remappings, including intrinsics such as transpositions.

A compilation technique was presented to handle HPF remappings efficiently. Useless remappings are removed. 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 were presented. Namely, a minimal number of messages containing only the required data is sent over the network. Thus the technique minimizes both latency and bandwith-related costs 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.

This technique is implemented in HPFC [7, 8, 9], a prototype HPF compiler developed within the PIPS project [20]. Portable PVM code is generated.

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 codes too tricky.

Future work includes:

- mappings code motion to reduce the number of executed remappings,
- reducing the remapped elements set to what is needed through advanced compile time analyses [11, 10].
- 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 François BODIN for informal discussions, Bill PUGH for suggestions, Béatrice CREUSILLET for pointers, Pierre JOUVELOT for corrections and François IRIGOIN for the improvements he suggested.

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] 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.

[5] 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.

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

[7] 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.

[8] 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. Also available as TR EMP A/257/CRI.

[9] Fabien Coelho. Compilation of I/O communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995. Also available as TR-CRI A/264.

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

[11] Béatrice Creusillet and François Irigoin. Interprocedural array regions analyses. In *Language and Compilers for Parallel Computing*, August 1995.

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] Ken Kennedy, Nenad Nedeljković, and Ajay Sethi. A linear time algorithm for computing the memory access sequence in data-parallel programs. CRPC-TR 94485-S, Center for Research on Parallel Computation, Rice University, October 1994. Submitted to PPoPP'95.

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

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

[26] 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.

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

[28] 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.

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

[30] 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.

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

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

[33] Vincent Van Dongen. Array redistribution by scanning polyhedra. Personnal communication: One page summary for PARCO'95, July 1995.

[34] 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.

[35] 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.

Appendix

This section details the linear constraints of Figure 7 that are derived from the running example presented in Figure 6.

The first two inequalities come from Array `A` declaration. Variables α define array dimensions. The next ten inequalities characterize template declarations whose dimensions are represented by θ variables. The processor declarations using ψ variables are define by the following ten constraints.

in HPF, array elements are aligned with the template cells through an affine expression. Each array index can be used at most once in a template subscript expression in any given alignment and each subscript expression cannot contain more that one index. Replication onto template cells, noted *, expresses the duplication of array elements along the replicated dimension of the template. Figure 6 illustrates 2 alignments where replication occures. `align a(i) with t(*,i,*)` expresses that array element `a(i)` is duplicated onto the first and third dimensions of template `t`. `align a(i) with t'(i,*)` duplicates array element `a(i)` onto the cells of the template row `i`. The alignment constraints simply link the array (α) to the template (θ) variables, following the affine alignment subscript expressions.

The distribution constraints introduce the block variables (δ), and the cyclic distribution on the first dimension of template `t'` requires a cycle variable (γ'_1) which counts the cycles over the processors for a given template cell. Each distributed dimension is described by an equation that links the template (θ) and the processor dimensions ψ together with the added variables (δ,γ). The distribution of replicated dimensions

(θ_1 and θ_3 for `t` and θ'_2 for `t'`) is not represented since it corresponds to extra and useless information. Since data are replicated, the block distribution information does not need to be taken into account.

Figure 10 illustrate the communications performed from source to target processors according to the redistribution of A from `T` to `T'`. The four source processors (*,1,*) (as well as (*,2,*)), share the same set of array elements to be sent. Any of them can transfer these elements to the target processors. Our code generation algorithm assigns themselves the target processor set in a cyclic way.