Compiling Dynamic Mappings with Array Copies
Report EMP CRI A-292

Fabien COELHO (coelho@cri.ensmp.fr)

Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, F-77305 Fontainebleau cedex, FRANCE
phone: +33 1 64 69 48 52

May 1996 (updated June 1996)

Abstract

Array remappings are useful to many applications on distributed memory parallel machines. They are integrated into High Performance Fortran, a Fortran 90-based data-parallel language. This paper describes techniques for handling dynamic mappings through simple array copies: array remappings are translated into copies between statically mapped distinct versions of the array. It discusses the language restrictions required to do so. The remapping graph which captures all remapping and liveness information is presented, as well as additional data-flow optimizations that can be performed on this graph, so as to avoid useless remappings at runtime. Such useless remappings appear for arrays that are not used after a remapping. Live array copies are also kept to avoid other flow-dependent useless remappings. Finally the code generation and runtime management issues needed by our scheme are discussed. These techniques are implemented in our prototype HPF compiler.

1 Introduction

Array remappings, *i.e.* the ability to change array mappings at runtime, are definitely useful to applications and kernels such as ADI[10], linear algebra solvers [12], 2D FFT[5] and signal processing [11] for efficient execution on distributed memory parallel computers. HPF [9] provides explicit remappings through `realign` and `redistribute` directives and implicit ones at subroutine calls and returns for array arguments. This paper discusses compiler handling of remappings and associated data flow optimizations.

1.1 Motivation

These features are costly at runtime because they imply communication. Moreover, even well written HPF programs may require useless remappings that can be avoided. In Figure 1 the change of both alignment and distribution of `A` requires two remappings while it could be remapped at once from `(block,*)`

```
!hpf$ align with B:: A
!hpf$ distribute B(block,*)
      ...
! A is remapped
!hpf$ realign A(i,j) with B(j,i)
! A is remapped again
!hpf$ redistribute B(cyclic,*)
```

Figure 1: Possible direct `A` remapping

```
!hpf$ align with B:: C
!hpf$ distribute B(block,*)
      ...
! C is remapped
!hpf$ realign C(i,j) with B(j,i)
! C is remapped back to initial!
!hpf$ redistribute B(*,block)
```

Figure 2: useless `C` remappings

```
!hpf$ align with T :: &
!hpf$     A,B,C,D,E
!hpf$ distribute T(block)
      ... A B C D E ...
!hpf$ redistribute T(cyclic)
      ... A D ...
```

Figure 3: Aligned array remappings

```
  real Y(1000)
!hpf$ distribute Y(block)
  interface
    subroutine foo(X)
      real X(1000)
!hpf$ distribute X(cyclic)
    end subroutine
    subroutine bla(X)
      real X(1000)
!hpf$ distribute X(cyclic(2))
    end subroutine
  end interface
  ... Y ...
  call foo(Y)
  call foo(Y)
  call bla(Y)
  ... Y ...
```

Figure 4: Useless argument remappings

to `(*,cyclic)` rather than using the intermediate `(*,block)` mapping. In Figure 2 both `C` remappings are useless because the redistribution restores its initial mapping. In Figure 3 template `T` redistribution enforces the remapping of all five aligned arrays, although only two of them are used afterwards. In Figure 4 the consecutive calls to subroutine `foo` remap the argument on entry in and on exit from the routine, and both back and forth remappings could be avoided between the two calls; moreover, between calls to `foo` and `bla`, array `Y` is remapped from `(cyclic)` to `(block)` and then from `(block)` to `(cyclic(2))` while a direct remapping would be possible. All these examples do not arise from badly written programs, but from a normal use of HPF features. They demonstrate the need for compile-time data flow optimizations to avoid remappings at runtime.

1.2 Related work

Such optimizations, especially the interprocedural ones, are discussed in Hall *et al.* [6]. For handling subroutine calls, they show that the right approach is

```
!hpf$ template T1,T2
!hpf$ align with T1 :: A
      ... A ...
      if (...) then
!hpf$   realign with T2 :: A
        ... A ...
      endif
!hpf$ redistribute T2
      ... A ...
```

```
!hpf$ distribute A(block)
      ... A ...
      if (...) then
!hpf$   redistribute A(cyclic)
        ... A ...
      endif
      ...
      ! no reference to A
      ...
!hpf$ redistribute A(cyclic(10))
      ... A ...
```

Figure 5: Ambiguity of remappings (a) and (b)

that the caller must comply to the callee requirements, that is the caller must insures that the arguments are properly mapped before the call. Our approach follows these lines. However, their analysis and optimization algorithms do not suit HPF for several technical reasons: first, they overlook the fact that in the general case several mappings can be associated at runtime to an array reference; second, they do not address the practical handling of array references; third, the two level mappings of HPF cannot be directly adapted to the simple algorithms described or suggested.

Let us focus on their work. *Reaching decompositions* are computed within procedures to describe possible mappings of array references:

> ([6] p. 525) To determine the decomposition of distributed arrays at each point in the program, the compiler calculates *reaching decompositions*. Locally, it is computed in the same manner as *reaching definitions*, with each decomposition treated as a "definition" [1].

Although it is indeed a dataflow problem, the computation of reaching mappings is not as simple as the reaching definitions problem because the compiler cannot directly know the mapping of arrays while looking at a remapping: whether an array is remapped or not may depend on the control flow, as shown in Figure 5(a); array `A` mapping is modified by the `redistribute` if the `realign` was executed before at runtime, otherwise `A` is aligned with template T_1 and get through T_2 redistribution unchanged; thus reaching mappings must be computed from the entry point of the program, propagating each initial mapping; however, unlike the reaching definition problem, new mappings (definitions) may appear in the graph as possible mappings reaching a remapping are changed by the directive.

Useless remappings are also removed by computing *live decompositions*:

> ([6] p. 529) We define *live decompositions* to be the set of data composition specifications that may reach some array reference aligned with the decomposition. [...] We calculate live decompositions by simply propagating uses backwards through the local control flow graph for each procedure [1].

The propagation is stopped by remappings that affects the given used array. However, here again, the ambiguity problem due to the two level mapping of

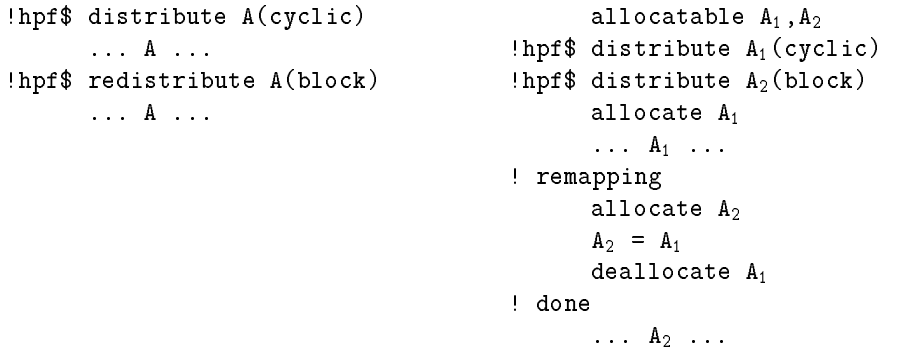

Figure 6: Translation from dynamic to static mappings

HPF makes this fact not obvious to the compiler: the propagation should be stopped or not depending on the control flow followed by the program.

Finally optimizations are suggested for coalescing *live decompositions*, extracting loop-invariant mappings and using array kill analysis to avoid actually remapping dead arrays. Their approach assumes that a collection of library routines is available to remap data as needed.

In contrast to this approach, our scheme handles array references explicitely (by substituting the appropriate array copy), and thus enables more subsequent optimizations. We build a graph representation of the remappings ($\mathcal{G}_R$) which associates source to target copies, plus a precise use information (array may be read and/or written). We then repropagate live mappings to associate live source and target mapping couples after having removed dead remappings. This recomputation extracts the liveness analysis benefit and also performs the coalescing of live mappings in a single optimization on the small $\mathcal{G}_R$ graph. A second compile-time optimization allows to keep maybe-live array copies so as to avoid more useless remappings: the runtime management traces the status of live copies that may depend of the control flow. The drawback is the language restrictions needed for doing so in the general case. These issues are discussed in the following.

1.3 Outline

This paper focuses on compiling HPF remappings with array copies. Required language restrictions are discussed: basically, the compiler must statically know the mapping of array references, so as to apply this approach. Under a few constraints a program with dynamic mappings can be translated into a standard HPF program with copies between differently mapped arrays, as outlined in Figure 6: the redistribution of array `A` is translated into a copy from $\mathtt{A}_1$ to $\mathtt{A}_2$; the array references are updated to the right array version.

By doing so, the benefit of remappings is limited to software engineering issues since it is equivalent to some static HPF program to be derived. It may also be argued that much expressiveness is lost by restricting the language. However it must be noted that: (1) software engineering is an issue that deserves consideration; (2) the current status of the language definition is to drop

remappings as a whole because they are considered too difficult to handle; (3) no real application we have encountered so far that would have benefit from remappings require the full expressiveness of arbitrary flow-dependent dynamic mappings. Thus it makes sense to suggest to keep the simple and interesting aspects of a feature and to delay further powerful extentions when both the need and compilation techniques are available.

Such an approach is also required if one wants to compile remappings rather than relying on generic library functions. Indeed, for compiling a remapping into a message-passing SPMD code [2] both source and target mapping couples must be known. Then the compiler can take advantage of all available information to generate efficient code. The paper also addresses associated data flow optimizations enabled by such a scheme, by removing useless remappings and keeping track of live array copies.

Section 2 discusses the language restrictions and the handling of subroutine mappings. Section 3 describes the building of the remapping graph which extract remapping and liveness information from the control flow graph. Section 4 then presents data flow optimizations that can be performed on this graph. Finally Section 5 discusses runtime implications of our compilation and optimization techniques, before concluding.

2 Overview

This paper aims at compiling explicit and implicit HPF remappings by translating them into runtime array copies. We first discuss the language restrictions needed to apply this scheme. Then the subroutine argument mapping translation into explicit array copies is outlined.

2.1 Language restrictions

In order to be able to translate dynamic mappings into differently distributed array copies, enough information must be available to the compiler. Basically, the compiler must statically know about array mappings for each array reference. Thus the following constraints are needed:

1. dynamic mappings must be as constrained as static mappings;

   (thus they cannot depend on runtime parameters)

2. there must be no control flow mapping ambiguities for a reference;

   (however there may be such an ambiguity at a point in the program provided that the array is not referenced: in Figure 5(b) after the `endif` and before the final redistribution the compiler cannot know whether array `A` is distributed block or cyclic, but the mapping ambiguity is solved before any reference to `A`)

3. explicit mapping interfaces for subroutine arguments are required;

Constraint 1 insures that differently distributed array copies can be declared a static mapping. Thus array remappings cannot depend on runtime values as currently allowed by HPF. Constraint 2 is required so as to use only one possible new array for each reference. This constraint will be checked by the

remapping graph construction algorithm which handles the substitution. Finally Constraint 3 must be met so that the caller knows about the callee requirements and is able to comply with them, as suggested in Section 2.2.

Under these constraints the compiler can switch all array references to different arrays depending on the array mapping at this point, and to insert array copies to update data when remappings are required. The expressiveness of array remappings is definitely reduced by these constraints, which restrict general dynamic array remappings to a static version. However it must be pointed out that no application or kernel we are aware of requires such expressiveness. Moreover although one can imagine some applications that would find some benefit from runtime dependent mappings, it is very hard to imagine codes that would both make sense from the application and performance point of view and that would present control flow-related mapping ambiguities.

2.2 Subroutine arguments handling

Subroutine arguments mappings will be handled as local remappings by the caller. Let us consider the different dummy arguments mappings the compiler may have to handle:

descriptive: when encountering a descriptive mapping for an array dummy argument at a call cite, the compiler will check that the current mapping of the actual array is compatible with this mapping.

prescriptive: the mapping is then enforced by the caller, by copying the array into an actual copy mapped as the corresponding dummy argument.

transcriptive: `inherit` tells that the callee can handle any mapping, thus the caller compilation is not constrained. However from the callee point of view handling inherited mapping array references is similar to handling remapping-related ambiguous references and is likely to be inefficient because all address computations must be delayed to run time.

Thus implicit remappings are translated into explicit ones at call site in the caller, provided that the dummy argument mapping is known to the caller through an appropriate explicit interface declaration. The intent attribute (`in`, `out` or `inout`) provides additionnal information about the effects of the call onto the array that will be used to determine live copies.

Under the proposed scheme we plan to respect the intended semantics of HPF argument passing, that is the argument is the only information the callee obtain from the caller. Thus explicit remappings of arguments within the callee will only affect copies local to the subroutine. Under more advance calling conventions, it may be thought of passing live copies along the required copy, so as to avoid further useless remappings within the subroutine.

3 Remapping graph $\mathcal{G}_R$

This section defines and describes the construction of the remapping graph. This graph is a subgraph of the control flow graph which captures remapping information such as the source and target copies for each remapping of an array

and how the array is used afterwards, that is a liveness information. Subsequent optimizations will be expressed on this small graph.

3.1 Definition and notation

In the following we will distinguish the abstract array and its possible instances with an associated mapping. Arrays are denoted by capital typewriter letters as `A`. Mapped arrays are associated a subscript such as $\mathtt{A}_2$. Differently subscripted arrays refer to differently mapped instances. A zero-subscripted copy $\mathtt{A}_0$ designates the initial static mapping of the array. $C(\mathtt{A})$ denote the set of all possible mappings of abstract array `A`. Through our compilation approach, each array and associated mapping will be given a new name, and copies between such arrays will model remappings.

If $\mathcal{G}$ is a graph then $\mathcal{V}(\mathcal{G})$ is its set of vertices and $\mathcal{E}(\mathcal{G})$ its set of edges. Successors of a vertex are designated by $\mathrm{succ}(v)$ and predecessors by $\mathrm{pred}(v)$.

Let us formally introduce the remapping graph $\mathcal{G}_R$. This graph is a subgraph of the control flow graph, and is expected to be much smaller in the general case. Our subsequent optimizations are expressed on this small graph instead of the control flow graph, avoiding many simple information propagations along paths.

vertices $\mathcal{V}(\mathcal{G}_R)$: the vertices are the remapping statements, whether explicit (`realign`, `redistribute`) or added to model implicit remappings at call sites; there is a subroutine entry point vertex v_0 and an exit point v_e.

edges $\mathcal{E}(\mathcal{G}_R)$: an edge denotes a possible path in the control flow graph with the same array remapped at both vertices.

labels: to each vertex v in the remapping graph is associated $S(v)$, the set of remapped arrays. For each array $\mathtt{A} \in S(v)$ we have:

- $L_{\mathtt{A}}(v)$**:** *one* (or none, noted $\perp$) leaving array copy, *i.e.* the copy which must be referenced after the remapping; note that in the general case HPF allows several leaving mappings as depicted in Figure 7: array `A` is remapped at the `redistribute` to `(block,*)` or `(*,block)` depending on the execution of the `realign`.

  We assume that no such cases occur to simplify our presentation.

- $R_{\mathtt{A}}(v)$**:** the set of reaching copies for the array at this vertex.

  In the general case with several leaving copies, distinct reaching copy sets must be associated to each possible leaving copy.

- $U_{\mathtt{A}}(v)$**:** how the leaving copy might be used afterwards. It may be not referenced (`N`), fully redefined before any use (`D`), only read (`R`) or modified (`W`). The use information qualifiers supersede one another in the given order, *i.e.* once a qualifier is assigned it can only be updated to a stronger qualifier. The default value is `N`.

  This provides a precise live information that will be used by the runtime and other optimizations to avoid remappings by detecting and keeping live copies. However it must be noted that this information is conservative, because abstracted at the high remapping graph level. The collected information can differ from the actual runtime effects on the subroutine: an array can be qualified as `W` from a point and not be actually modified.

```
!hpf$ distribute T(*,block)
!hpf$ align A(i,j) with T(i,j)
      if (...) then
!hpf$   realign A(i,j) with T(j,i)
      endif
!hpf$ redistribute T(block,*)
```

Figure 7: Several leaving mappings

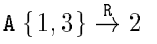

Figure 8: Label representation

Each edge is labelled with the arrays that are remapped from at the sink vertex when coming from the source vertex: $A(v, v')$. Note that

$$\mathtt{A} \in A(v, v') \Rightarrow \mathtt{A} \in S(v) \text{ and } \mathtt{A} \in S(v')$$

Figure 8 shows a label representation. Array `A` remapping links reaching copies $\{1, 3\}$ to the leaving mapping 2, the new copy being only read (`R`). The information means that the vertex is a remapping for array `A`. It may be reached with copies $\mathtt{A}_1$ and $\mathtt{A}_3$ and must be leaved with copy $\mathtt{A}_2$. As this copy will only be read, the compiler and runtime can keep the reaching copy values which are live, as they may be useful for instance if the array is to be remapped back to one of these distributions.

3.2 Construction algorithm

The remapping graph described above holds all the remapping and liveness information. The next issue is to build the graph. This section outlines the construction algorithm, which builds the remapping graph and updates the control graph to (1) switch array references to the appropriate copy, distributed as expressed by the program, (2) reflect implicit remappings of array arguments through explicit remappings and (3) check the conditions required for the correctness of our scheme.

The remapping graph construction algorithm starts by propagating the initial mapping copy of the array from the entry point of the subroutine. $\mathcal{G}_R$ construction algorithm pushes array versions along the control graph and extract a simpler graph to reflect the needed runtime copies to comply to the intended semantics of the program. Two added vertices (call v_c and exit v_e) model the dummy arguments imported and/or exported values.

Here is such a data flow formulation of the construction algorithm. First, let us define the sets that will be computed by the dataflow algorithms in order to build $\mathcal{G}_R$:

REACHING(v): the set of arrays and associated mappings reaching vertex v; these arrays may be remapped at the vertex or left unchanged, thus going through the vertex.

LEAVING(v): the set of arrays and associated mappings leaving vertex v; one leaving mapping per array is assumed for simplifying the presentation.

REMAPPED(v): the set of arrays actually remapped at vertex v. (note that if several leaving array mappings are allowed, this information is associated to array and mapping couples instead of just considering arrays).

EFFECTSOF(v): the proper effect on distributed variables of vertex v, *i.e.* these variables and whether they are not referenced, fully redefined, defined or used; this is assume as an available information.

EFFECTSAFTER(v): the distributed variables and associated effects that may be encountered after v and before any remapping of these variables.

EFFECTSFROM(v): just the same, but including also the effects of v.

REMAPPEDAFTER(v): the distributed variables and associated remapping vertices that may be encountered directly (without intermediate remapping) after v.

REMAPPEDFROM(v): just the same, but including also v.

The following function computes the leaving mapping from a reaching mapping at a given vertex:

$\mathtt{A_j}$ = IMPACT($\mathtt{A_i}, v$): the resulting mapping of $\mathtt{A}$ after v when reached by $\mathtt{A_i}$; for all but remapping vertices, $\mathtt{A_i} = \mathtt{A_j}$, *i.e.* the mapping is not changed; realignments of $\mathtt{A}$ or redistributions of the template $\mathtt{A_i}$ is aligned with may give a new mapping. The impact of a call is null.

ARRAY($\mathtt{A_i}$)=$\mathtt{A}$: the function returns the array from and array and associated mapping.

operator −: to be understand as *but those concerning*, that is the operator is not necessarily used with sets of the same type.

Now, here is the construction algorithm expressed as a set of data flow equations.

Input:

- control flow graph $\mathcal{G}_C$ with entry v_0 and exit v_e vertices
- the set of remapping vertices V_R, which includes v_0 and v_e.
- the proper effects of vertices on distributed variables EFFECTSOF(v) (the default for V_R is no effects)
- for any remapped array at a vertex, there is only one possible leaving mapping. This assumption simplifies the presentation, but could be removed by associating remapped information to array mappings instead of the array.

Updating $\mathcal{G}_C$ (arguments): first let us update $\mathcal{G}_C$ to model the desired mapping of arguments.

- Add call vertex v_c and an edge from v_c to v_0 in $\mathcal{G}_C$.

intent	$U_{\mathtt{A}}(v_{\mathtt{c}})$	$U_{\mathtt{A}}(v_{\mathtt{e}})$
in	D	N
inout	D	W
out	N	W

Figure 9: Array argument use

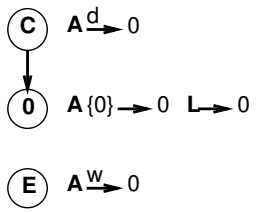

Figure 10: Initial $\mathcal{G}_R$

Reaching and Leaving mappings: are computed starting from the entry point in the program. Propagated mappings are modified by remapping statements as modeled by the IMPACT function, leading to new array versions to be propagated along $\mathcal{G}_C$. This propagation is a may forward dataflow problem.

initialization:

- $\textsc{Reaching} = \emptyset$
- $\textsc{Leaving} = \emptyset$
- add all argument distributed variables and their associated mappings to $\textsc{Leaving}(v_{\mathtt{c}})$ and $\textsc{Leaving}(v_{\mathtt{e}})$.
- update $\textsc{EffectsOf}(v_{\mathtt{c}})$ and $\textsc{EffectsOf}(v_{\mathtt{e}})$ as suggested in Figure 9: If values are imported the array is annotated as defined before the entry point. If values are exported, it is annotated as used after exit. This models safely the caller context the callee must assume to comply to the intended semantics.
- add all local distributed variables and their associated initial mapping to $\textsc{Leaving}(v_0)$.

Figure 10 shows the initial remapping graph with an `inout` intended array argument `A` and a local array `L`.

propagation:

- the array mappings reaching a vertex are those leaving its predecessors.
  $\textsc{Reaching}(v) = \bigcup_{v' \in \mathrm{pred}(v)} \textsc{Leaving}(v')$
- the array mappings leaving a vertex are updated with the impact of the statement on the array mappings reaching this vertex.
  $\textsc{Leaving}(v) = \textsc{Leaving}(v) \cup \bigcup_{a \in \textsc{Reaching}(v)} \textsc{impact}(a, v)$

Updating references for all vertices $v \in \mathcal{V}(\mathcal{G}_C) - V_R$ so that $\textsc{EffectsOf}(v)$ on is not `N`:

- if $|\{m \in \textsc{Leaving}(v), \textsc{array}(m) = \mathtt{A}\}| > 1$ then issues an error, because there is more than one mapping for a given array
- else substitute the references with the corresponding array copy.
- note that there may be none if some piece of code is dead.

Remapped arrays are directly extracted from REACHING; they are those transformed by IMPACT.

$\textsc{Remapped}(v) = \bigcup_{m \in \textsc{Reaching}(v), m \neq \textsc{impact}(m,v)} \textsc{array}(m)$

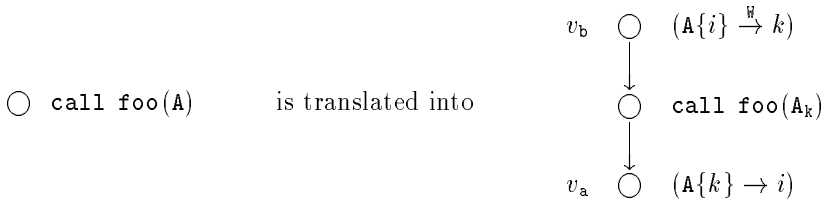

Figure 11: Call with a prescriptive `inout`-intended argument

Updating $\mathcal{G}_C$ (calls):

- calls with distributed arguments are managed as shown in Figure 11:
  $\text{pred}(v_b) = \text{pred}(v), \text{succ}(v_b) = \{v\}, \text{pred}(v_a) = \{v\}, \text{succ}(v_a) = \text{succ}(v), \text{pred}(v) = \{v_b\}, \text{succ}(v) = \{v_a\}$
  $\textsc{Remapped}(v_b) = \{A\}$
- V_R is updated accordingly: $V_R = V_R \cup \{v_b, v_a\}$

Summarizing effects: this phase summarizes the use information after remapping statements, and up to any other remapping statement. Hence it captures what may be done with the array mapping considered.

This phase is based proper effects that are directly extracted from the source code for direct references, or through intent declarations in subroutine explicit interfaces. Depending on the intent attribute to a subroutine argument the corresponding effect is described in Figure 12.

intent	effect
`in`	R
`inout`	W
`out`	D

Figure 12: Intent effect

Remapping statements but v_c and v_e have no proper effects:

$$\forall v \in V_R - \{v_c, v_e\}, \textsc{EffectsOf}(v) = \emptyset$$

This is a may backwards dataflow problem.

initialization: no effects!

- $\textsc{EffectsAfter} = \emptyset$
- $\textsc{EffectsFrom} = \emptyset$

propagation:

- the effects leaving a vertex are those from its successors.
  $\textsc{EffectsAfter}(v) = \bigcup_{v' \in \text{succ}(v)} \textsc{EffectsFrom}(v')$
- the effects from a vertex are those leaving the vertex and proper to the vertex, but remapped arrays.
  $\textsc{EffectsFrom}(v) = (\textsc{EffectsAfter}(v) \cup \textsc{EffectsOf}(v)) - \textsc{Remapped}(v)$

Computing $\mathcal{G}_R$ edges: as we expect few remappings to appear within a typical subroutine, we designed the remapping graph over the control graph with accelerating edges that will be used to propagate remapping information and optimization quickly. This phase propagates for once remapping statements (array and vertex couples) so that each remapping statement will know it possible successors for a given array.

This is a may backwards dataflow problem.

initialization:

- $\textsc{RemappedAfter} = \emptyset$
- initial mapping vertex couples are defined for remapping statement vertices and arrays remapped at this very vertex. $\textsc{RemappedFrom}(v) = \bigcup_{a \in \textsc{Remapped}(v)} \{(a, v)\}$

propagation:

- the remapping statements after a vertex are those from its successors. $\textsc{RemappedAfter}(v) = \bigcup_{v' \in \mathrm{succ}(v)} \textsc{RemappedFrom}(v')$
- the remapping statements from a vertex are updated with those after the vertex, but those actually remapped at the vertex. $\textsc{RemappedFrom}(v) = \textsc{RemappedFrom}(v) \cup (\textsc{RemappedAfter}(v) - \textsc{Remapped}(v))$

Generating $\mathcal{G}_R$: from these sets we can derive the remapping graph:

- V_R are $\mathcal{G}_R$ vertices
- edges and labels are deduced from RemappedAfter
- $S()$, $R()$ and $L()$ from Remapped, Reaching and Leaving
- $U()$ from EffectsAfter

All the computations are simple standard data flow problems, but the reaching and leaving mapping propagation: indeed, the IMPACT function may create new array mappings to be propagated from the vertex. The worst case complexity of the propagation and remapping graph algorithm describe above can be computed. Let us denote n is the number of vertices in $\mathcal{G}_C$, s the maximum number of predecessors or successors of a vertex in $\mathcal{G}_C$, m the number of remapping statements (including the entry and exit points), p the number of distributed arrays. With the simplifying assumption that only one mapping may leave a remapping vertex, then the maximum number of mappings to propagate is mp. Each of these may have to be propagated through at most n vertices with a smp worst case complexity for a basic implementation of the union operations. Thus we can bound the worst case complexity of the propagation to $\mathcal{O}(nsm^2p^2)$.

3.3 Example

Let us focus on the example in Figure 13. The small subroutine contains four remappings, thus with the added call, entry and exit vertices there are seven vertices in the corresponding remapping graph. There are three arrays, two of which are local and the last one which is passed as an argument to the subroutine. The sequential loop structure with two remappings is typical of

```
  subroutine remap(A,m)                 → C
    parameter(n=1000)
    intent(inout):: A
    real, dimension(n,n):: A,B,C
!hpf$ align with A:: B,C
!hpf$ distribute * A(block,*)           → 0
    ... B written, A read
    if (...B read) then
!hpf$ redistribute A(cyclic,*)          → 1
      ... A p written, A B read
    else
!hpf$ redistribute A(block,block)       → 2
      ... p written, A read
    endif
    do i=1, m+p
!hpf$ redistribute A(block,*)           → 3
      ... C written, A read
!hpf$ redistribute A(*,block)           → 4
      ... A written, A C read
    enddo
  end subroutine remap                  → E
```

Figure 13: Code example

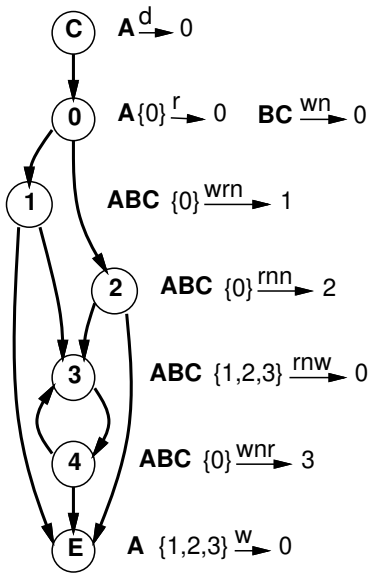

Figure 14: Remapping graph

ADI. The ... notation stands for any piece of code with the specified effects on the variables.

Figure 14 shows the resulting remapping graph and the associated liveness information for each array represented above the arrow. The rationale for the 1 to E and 2 to E edges is that the loop nest may have no iteration at runtime, thus the remappings within the array may be skipped. Since all arrays are aligned together, they are all affected by the remapping statements. Four different versions of each array might be needed with respect to the required different mapping. However, the liveness analysis shows that some instances are never referenced such as $\mathtt{B}_3$ and $\mathtt{C}_1$.

4 Data flow optimizations

The remapping graph $\mathcal{G}_R$ constructed above abstracts all the liveness and remapping information extracted from the control flow graph and the required dynamic mapping specifications. Following [6] we plan to exploit as much as possible this information to remove useless remappings that can be detected at compile time, or even some that may occur under particular runtime conditions. These optimizations on $\mathcal{G}_R$ are expressed as standard data flow problems [8, 7, 1].

4.1 Removing useless remappings

Leaving copies that are not live appear in $\mathcal{G}_R$ with the `N` (not used) label. It means that although some remapping on an array was required by the user, this array is not referenced afterwards in its new mapping. Thus the copy update

is not needed and can be skipped. However, by doing so, the set of copies that may reach latter vertices is changed. Indeed, the whole set of reaching mappings must be recomputed. It is required to update this set because we plan a compilation of remappings, thus the compiler must know all possible source and target mapping couples that may occur at run time. This recomputation is a may forward standard data-flow problem.

This optimization is performed as follows:

remove useless remappings:

simply by deleting the leaving mapping of such arrays.

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), U_{\mathtt{A}}(v) = \mathtt{N} \Rightarrow L_{\mathtt{A}}(v) = \bot$$

recompute reaching mappings:

initialization: use 1-step reaching mappings

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), R_{\mathtt{A}}(v) = \bigcup_{\substack{v' \in \mathrm{pred}(v) \\ \mathtt{A} \in A(v',v), U_{\mathtt{A}}(v') \neq \mathtt{N}}} L_{\mathtt{A}}(v')$$

Reaching mappings at a vertex are initialized as the leaving mappings of its predecessors which are actually referenced.

propagation: optimizing function

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), R_{\mathtt{A}}(v) = R_{\mathtt{A}}(v) \cup \bigcup_{\substack{v' \in \mathrm{pred}(v) \\ \mathtt{A} \in A(v',v), U_{\mathtt{A}}(v') = \mathtt{N}}} R_{\mathtt{A}}(v')$$

The function propagates reaching mappings along paths on which the array is not referenced, computing the transitive closure of mappings on those paths.

The iterative resolution of the optimizing function is increasing and bounded, thus it converges.

Let us assume $\mathcal{O}(1)$ set element put, get and membering functions. 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 $\mathcal{O}(m^2pqr)$. Note that m, q and r are expected to be very small.

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):*

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), U_{\mathtt{A}}(v), \forall a \in R_{\mathtt{A}}(v),$$
$$\exists v' \textit{ and a path from } v' \textit{ to } v \textit{ in } \mathcal{G}_R,$$
$$\textit{so that } a \in L_{\mathtt{A}}(v') \textit{ and } \mathtt{A} \textit{ is not used on the path.}$$

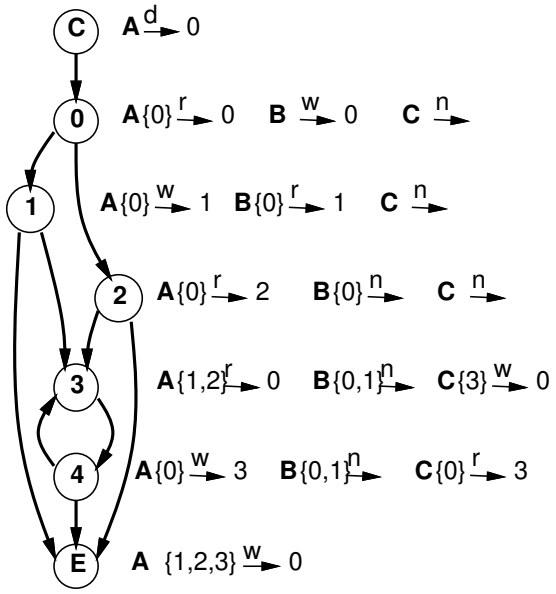

Figure 15: Example after optimization

Proof sketch: 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.

Figure 15 displays the remapping graph of our example after optimization. From the graph it can be deduced that array `A` may be used with all possible mappings $\{0,1,2,3\}$, but array `B` is only used with $\{0,1\}$ and array `C` with $\{0,3\}$. Array `C` is not live but within the loop nest, thus its instantiation can be delayed, and may never occur if the loop body is never executed. Array `B` is only used at the beginning of the program, hence all copies can be deleted before the loop. The generation of the code from this graph is detailed in Section 5.

4.2 Dynamic live copies

Through the remapping graph construction algorithm, array references in the control graph $\mathcal{G}_C$ were updated to an array version with a statically known mapping. The remapping graph holds the information necessary to organize the copies between these versions in order to respect the intended semantics of the program. The first idea is to allocate the leaving array version when required, to perform the copy and to deallocate the reaching version afterwards.

However, some copies could be kept so as to avoid useless remappings when copying back to one of these copies if the array was only read in between. The remapping graph holds the necessary information for such a technique. Let us consider the example in Figure 16 and its corresponding remapping graph in Figure 17. Array `A` is remapped differently in the branches of the condition. It may be only modified in the `then` branch. Thus, depending on the execution path in the program, array copy $\mathtt{A}_0$ may reach remapping statement 3 live or not. In order to catch such cases, the liveness management is delayed until run time:

```
!hpf$ distribute A(block)            → 0
      ... A read
      if (...) then
!hpf$   redistribute A(cyclic)       → 1
        ... A written
      else
!hpf$   redistribute A(cyclic(2))    → 2
        ... A read
      endif
!hpf$ redistribute A(block)          → 3
      ... A read
      end
```

Figure 16: Flow dependent live copy

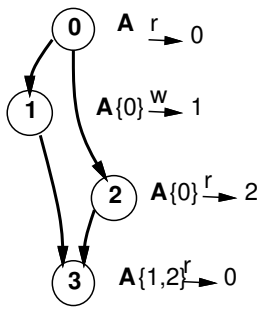

Figure 17: Corresponding $\mathcal{G}_R$

dead copies will be deleted (or mark as dead) at the remapping statements.

For the example above, when executing remapping statement 1 the runtime knows that the array may be modified, thus can tag copy 0 as dead, but when executing remapping statement 2 the array is only read afterwards hence the copy can be kept as live. At remapping statement 3 the runtime can then check for the liveness tag associated to array version 0 and avoid a copy and the associated communication if live. Thus simple runtime data structures can keep track of live copies and avoid remappings, depending on the execution path of the program.

Keeping array copies so as to avoid remappings is a nice but expensive optimization, because of the required memory. Thus it would be interesting to keep only copies that may be used latter on. In the example above it is useless to keep copies $\mathtt{A}_1$ or $\mathtt{A}_2$ after remapping statement 3 because the array will never be remapped to one of these distribution. Determining at each vertex the set of copies that may be live and used latter on is a may backward standard data flow problem: leaving copies must be propagated backward on paths where they are only read. Let $M_{\mathtt{A}}(v)$ be the set of copies that may be live after v.

initialization: directly useful mappings

$$\forall v \in \mathcal{V}(\mathcal{G}_R), \forall \mathtt{A} \in S(v), M_{\mathtt{A}}(v) = L_{\mathtt{A}}(v)$$

propagation: optimizing function

$$\forall v \in \mathcal{V}(\mathcal{G}_R),\ \forall \mathtt{A} \in S(v),\ U_{\mathtt{A}}(v) \in \{\mathtt{N}, \mathtt{R}\}, M_{\mathtt{A}}(v) = M_{\mathtt{A}}(v) \cup \bigcup_{\substack{v' \in \mathrm{succ}(v) \\ \mathtt{A} \in A(v,v')}} M_{\mathtt{A}}(v')$$

Copies that may be of interest are propagated towards predecessors if the array is not modified (neither `W` nor `D`).

At a remapping statement v the runtime should only keep the live copies described in $M(v)$. Moreover, as the liveness information is managed dynamically, the runtime may decide to free some live copies because not enough memory is available. This would simply lead to a copy based update of the values of the array instead of the communication-free reuse of the values if the copy had been kept.

```
!hpf$ distribute A(block)
      ... A
      do i=1, t
!hpf$   redistribute A(cyclic)
        ... A
!hpf$   redistribute A(block)
      enddo
      ... A
```

Figure 18: Loop invariant remappings

```
!hpf$ distribute A(block)
      ... A
      do i=1, t
!hpf$   redistribute A(cyclic)
        ... A
      enddo
!hpf$ redistribute A(block)
      ... A
```

Figure 19: Optimized version

Such a scheme makes an implicit choice of granularity for managing live copies, by staying at the remapping statement level. One could imagine a very precise runtime tagging of modified arrays. However, where to put the needed tag updates so as to avoid any significant runtime overhead is not obvious. At the remapping graph level, communication is expected to occur at vertices, thus the runtime implication of the liveness management should be very small compared to these costs. Moreover it is likely that the remappings have been carefully placed in the program.

4.3 Other optimizations

Further optimization can be thought of, as discussed in [6].

Array kill analysis, for instance based on array regions [4], tells whether the values of an array are dead at a given point in the program. This semantical analysis can be used to avoid remapping communication of values that will never be reused. Array regions can also describe a subset of values which are live, thus the remapping communication could be restricted to these values, reducing communication costs further. However such compile-time intensive advanced semantical analyses are not the common lot of commercial compilers. Our prototype HPF compiler includes a `kill` directive for the user to provide this information. The directive simply creates a remapping vertex with the `D` use information, which will require no communication as discussed in Section 5.

Another idea is to move remappings around in the control flow graph, especially for moving them out of loops. However some pitfalls must be avoided. From the code in Figure 18 we suggest to move the remappings as shown in Figure 19. This differs from [6]: the initial remapping is not moved out of the loop, because if $\texttt{t} < 1$ this would induce a useless remapping. The remapping from `block` to `cyclic` will only occur at the first iteration of the loop. At others, the runtime will notice that the array is already mapped as required just by an inexpensive check of its status.

5 Runtime issues

The remapping graph information describying array versions reaching and leaving remapping vertices must be embedded into the program through actual copies in order to fullfil the semantical requirements for the correctness of the execution. Moreover some optimizations described in the previous sections rely

```
      interface
        subroutine foo(X)
!hpf$     distribute X(block)
        end subroutine
      end interface
!hpf$ distribute A(cyclic)
      ... A
      if (...) then
!hpf$   redistribute A(cyclic(2))
        ... A
      endif
! A is cyclic or cyclic(2)
! foo requires a remapping
      call foo(A)
```

Figure 20: Subroutine calls

```
! save the reaching status
      reaching(A)=status(A)
!hpf$ redistribute A(block)
      call foo(A)
! restore the reaching mapping
      if (reaching(A)=1) then
!hpf$   redistribute A(cyclic)
      elif (reaching(A)=2) then
!hpf$   redistribute A(cyclic(2))
      endif
```

Figure 21: Mapping restored

on the runtime to be performed. This section presents the runtime implications of our scheme.

5.1 Runtime status information

First, some data structure must be managed at run time to store the needed information: namely, the current status of the array (which array version is the current one and may be referenced); the live copies; the reaching status at a call cite.

The current status of an array can be simply described by a descriptor holding the version number, for instance `status(A)` for array `A`. By testing this status, the runtime will be able to notice which version of an array reaches a remapping statement. This status will have to be updated each time the array is remapped.

Secondly, the runtime must be able to test whether a version of a given array is live at a point. This is a boolean information to be attached to each array version. It will be denoted `live(`A_x`)` for liveness of array A_x. The liveness information will be updated at each remapping vertex, depending of the latter use of the copies from this vertex.

If interpreted strongly, Constraint 2 (p. 5) may imply that arrays as call arguments are considered as references and thus should not bare any ambiguity, such as the one depicted in Figure 20. However, since an explicit remapping of the array is inserted, the ambiguity is solved before the call, hence there is no need to forbid such cases. The issue is to restore the appropriate reaching mapping on return from the call. This can be achieved by saving the current status of the array that reached the call as suggested in Figure 21. Variable `reaching(A)` holds the information. The saved status is then used to restore the initial mapping after the call.

5.2 Copy code generation

The algorithm for generating the copy update code and live management from the remapping graph is outlined in Figure 23. Copy allocation and deallocation

```
if (status(A)≠2) then
  allocate A2 if needed
  if (not live(A2)) then
    if (status(A)=1) A2=A1
    if (status(A)=3) A2=A3
    live(A2)=true
  endif
  status(A)=2
endif
```

Figure 22: Code for Figure 8

are inserted in the control flow graph to perform the required remappings. The algorithm relies on the sets computed at the $\mathcal{G}_R$ optimization phase to insert only required copies and to keep only live copies that may be useful while others are cleaned.

The first loop inserts the runtime data structure management initialization at the entry point in the subroutine. All copies are denoted as not live, and no copy receives an *a priori* instantiation as the current copy. The rationale for doing so is to delay this instantiation to the actual use of the array, which is not necessarily with the initial mapping, and will not be necessarily used, as Array `C` in Figure 13.

The second loop nest extracts from the remapping graph the required copy, for all vertex and all remapped arrays, if there is some leaving mapping for this array at this point. The generated code first checks whether the status is different from the required leaving status, in order to skip the following if the right copy is already ok. If not, it is first allocated. If the array is not to be fully redefined afterwards, a copy from the reaching copy to the desired leaving copy if performed if the values are not live. Then the current status of the array and its associated liveness information are updated.

Copies that were live before but that are not live any more are cleaned, *i.e.* both freed and marked as dead. Finally a full cleaning of all local arrays is inserted at the exit vertices. Figure 22 shows some example of generated copy code for the remapping vertex depicted in Figure 8, assuming that all copies may be useful for latter remappings.

It must be noted that dead arrays (`D`) do not require any actual array copy, thus none is generated, hence this will result in no communication at run time. Moreover, there is no initial mapping imposed from entry in the subroutine. If an array is remapped before any use, it will be instantiated at the first remapping statement encountered at runtime with a non empty leaving copy, which may be the entry vertex v_0 or any other. Finally, care must be taken not to free the array dummy argument copy, since it is referenced by the caller. Thus the `allocate` and `free` must be only applied to local variables.

Another benefit from this dynamic live mapping management is that the runtime can decide to free a live copy if not enough memory is available one the machine to keep it, and to change the corresponding liveness status. If required latter on, the copy will be regenerated, *i.e.* both allocated and properly initialized with communication. Since the generated code does not assume that any

live copy must reach a point in the program, but rather decided at remapping statements what can be done, the code for the communication will be available.

6 Conclusion

Most of the techniques described in this paper are implemented in our prototype HPF compiler. The standard statically mapped HPF code generated is then compiled, with a special code generation phase for handling remapping communication due to the explicit array copies. The described techniques, especially the runtime live management, could be extended to interprocedural copies with some conventions at call cites.

Acknowledgment

Thanks to Corinne Ancourt, Béatrice Creusillet, François Irigoin and Pierre Jouvelot for their helpful comments and suggestions.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

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

[3] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Verlag, August 1995.

[4] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming special issue on LCPC*, 24(6):?, 1996. Extended version of [3].

[5] 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.

[6] Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines. In *Supercomputing*, pages 522–534, 1992.

[7] 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.

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

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

```
for A ∈ S(v_0)
    append to v_0 "status(A)=⊥"
    for a ∈ C(A)
        append to v_0 "live(A_a)=false"
    end for
end for
for v ∈ V(G_R) − {v_c}
    for A ∈ S(v)
        if (L_A(v) ≠⊥) then
            append to v "if (status(A)≠ L_A(v)) then"
            append to v "allocate A_{L_A(v)} if needed"
            append to v "if (not live(A_{L_A(v)})) then"
            if (U_A(v) ≠ D) then
                for a ∈ R_A(v) − {L_A(v)}
                    append to v "if (status(A)=a) A_{L_A(v)}=A_a"
                end for
            end if
            append to v "live(A_{L_A(v)})=true"
            append to v "endif"
            append to v "status(A)=L_A(v)"
            append to v "endif"
        end if
        for a ∈ C(A) − M_A(v)
            append to v "if (live(A_a)) then"
            append to v "  free A_a  if needed"
            append to v "  live(A_a)=false"
            append to v "endif"
        end for
    end for
end for
for all A
    for a ∈ C(A)
        append to v_e "if (live(A_a) and needed) free A_a"
    end for
end for all
```

Figure 23: Copy code generation algorithm

[10] Ulrich Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, October 1995. Available as CRPC-TR95-559-S.

[11] Peter G. Meisl, Mabo R. Ito, and Ian G. Cumming. Parallel synthetic aperture radar processing on workstation networks. In *International Parallel Processing Symposium*, pages 716–723, April 1996.

[12] Loic Prylli and Bernard Tourancheau. Efficient block cyclic data redistribution. Research Report 2766, INRIA, January 1996. To appear in Europar'96.