Compiling Dynamic Mappings with Array Copies (TR EMP CRI A 302. To appear in PPoPP'97)

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|0) 1 64 69 48 52, fax: (+33|0) 1 64 69 47 09
URL: http://www.cri.ensmp.fr/pips

Abstract

Array remappings are useful to many applications on distributed memory parallel machines. They are available in High Performance Fortran, a Fortran-based data-parallel language. This paper describes techniques to handle 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 run time. 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 required 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 [16], linear algebra solvers [19], 2D FFT [10], signal processing [17] or tensor computations [2] for efficient execution on distributed memory parallel computers. HPF [15, 12] 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.

```
!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

1.1 Motivation

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

```
  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

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

Figure 5: Ambiguity of remappings

1.2 Related work

Such optimizations to avoid useless remapping communications, especially interprocedural ones, have been discussed [11, 18]. It is shown [11] that the best approach to handle subroutine calls is that callers must comply to callee requirements. We follow this approach. In contrast to these papers we rely on standard explicit interfaces to provide the needed information about callees while enabling similar optimizations. We do not expect real applications to provide many remapping optimization opportunities at the interprocedural level. Moreover requiring mandatory interprocedural compilation is not in the spirit of the HPF specification. Also, the techniques presented in these papers cannot be extended *directly* to HPF because HPF two-level mapping makes the *reaching mapping* problem not as simple as the *reaching definition* problem. Both the alignment and distribution problems must be solved to extract actual mappings associated to arrays in the program.

The *Static Distribution Assignment* scheme [18] to handle dynamic array references is very similar to our approach which uses distinct copies for each array mapping. Both schemes have been developed concurrently. Such techniques require *well behaved* programs: remappings should not appear anywhere in the program, to avoid references with ambiguous mappings as shown in Figure 5. We go a step further by suggesting [4] that the language should forbid such cases. This is supported by our experience with real applications that require dynamic mappings: ambiguous mappings are rather bugs to be reported.

Our approach is unique from several points. First, our optimizations are expressed on the remapping graph which captures all mapping and use information for a routine. This graph can be seen as the dual of a contracted control-flow graph as noted in [18]. The advantage is that our graph is much smaller than the usual control-flow graph. Second, read and write uses of arrays are distinguished, enabling the detection of *live* copies that can be reused without communication in case of a remapping. Third, our runtime can handle arrays with an ambiguous mapping, provided that it is not referenced in such a state. This requirement is weaker than the one for *well behaved* programs [18, 11], since it enables cases such as Figure 6.

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

Figure 6: Other ambiguity of remappings

1.3 Outline

This paper describes a practical approach to handle HPF remappings. All issues are addressed: languages restrictions (or corrections) required for this scheme to be applicable, actual management of simple references in the code, data-flow optimizations, down to the runtime system requirements. This technique is implemented in our HPF compiler [3].

First, Section 2 presents the language restrictions, the handling of subroutine calls and our general approach to compile remappings. Second, Section 3 focuses on the definition and construction of the remapping graph which captures all necessary remapping and liveness information on a contracted control flow graph. Third, Section 4 discusses data-flow optimizations performed on this small graph. These optimizations remove all useless remappings and detect live or may-be-live copies to avoid further communication. Finally, Section 5 outlines runtime requirements implied by our technique, before concluding.

2 Overview

This paper focuses on compiling HPF remappings with array copies and on suggesting optimization techniques to avoid useless remappings. The idea is to translate a program with dynamic mappings into a standard HPF program with copies between differently mapped arrays, as outlined in Figure 7: the redistribution of array A is translated into a copy from A_1 to A_2; the array references are updated to the appropriate array version.

2.1 Language restrictions

In order to do so, the compiler must know statically about mappings associated to every array references. Thus the

```
!
! dynamic mappings
!
!hpf$ distribute A(cyclic)
      ... A ...
!hpf$ redistribute A(block)
      ... A ...
```

```
!
! static mappings
!
      allocatable A1,A2
!hpf$ distribute A1(cyclic)
!hpf$ distribute A2(block)
      allocate A1
      ... A1 ...
! remapping
      allocate A2
      A2 = A1
      deallocate A1
! done
      ... A2 ...
```

Figure 7: Translation from dynamic to static mappings

HPF language must be restricted to enable the minimum static knowledge required to apply this scheme. Namely:

1. References with ambiguous mappings due to the control-flow of the program are forbidden. Hence the compiler can figure out the mapping of array references and substitute the right copy.

2. Interfaces describing mappings of arguments of called subroutines are mandatory. Thus all necessary information is available for the caller to comply to the argument mapping of its callees.

3. *Transcriptive* mappings associated to subroutine arguments are forbidden. This feature can be replaced by a more precise mapping descriptions [6], or could be enabled but would then require an interprocedural compilation such as cloning [18].

Condition 1 is illustrated in Figure 5: 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. However there may be an ambiguity at a point in the program if the array is not referenced: in Figure 6 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`.

With these language restrictions the benefit of remappings is limited to software engineering issues since it is equivalent to a static HPF program. It may also be argued that 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 (by moving them out of the core language as simple approved extensions [12]) because they are considered too difficult to handle; (3) we have not encountered any real application so far that would benefit from the full expressiveness of arbitrary flow-dependent remappings. Thus

```
!
! implicit remapping
!
      interface
        subroutine CALLEE(A)
          intent(in), real:: A(1000)
!hpf$     distribute A(block)
        end subroutine
      end interface

      real B(1000)
!hpf$ distribute B(cyclic)

      ...
      call CALLEE(B)
      ...
```

```
!
! explicit remapping
!
      real B(1000)
!hpf$ dynamic B
!hpf$ distribute B(cyclic)

      ...
!hpf$ redistribute B(block)
! liveness: B is read
      call CALLEE(B)
!hpf$ redistribute B(cyclic)
      ...
```

Figure 8: Translation of a subroutine call

it makes sense to keep the simple and interesting aspects of remappings. Further powerful extensions can be delayed until applications need them and when compilation techniques are proven practical and efficient.

These language restrictions are also required to compile remappings rather than to rely on generic library functions. Indeed, for compiling a remapping into a message passing SPMD code [5] both source and target mappings must be known. Then the compiler can take advantage of all available information to generate efficient code. The implicit philosophy is that the compiler handles most of the issues at compile time, with minimum left to run time. But the language must require the user to provide the necessary information to the compiler. If not, only runtime-oriented approaches are possible, reducing the implementor's choices, but also performances.

2.2 Subroutine arguments

Subroutine argument mappings will be handled as local remappings by the caller. This is possible if the caller knows about the mapping required by callee dummy arguments, hence the above constraint to require interfaces describing argument mappings. The `intent` attribute (`in`, `out` or `inout`) provides additionnal information about the effects of the call onto the array. It will be used to determine live copies over call sites without interprocedural techniques.

Subroutine calls are translated as explicit remappings in the caller as suggested in Figure 8. Our scheme respects the intended semantics of HPF argument passing: the argument is the only information the callee obtains 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.

2.3 Discussion

The current HPF specification includes features (`inherit` directive for transcriptive mappings, possible ambiguous mappings, etc.) that make the runtime approach mandatory, at least for handling all cases. Another side-effect of optional interfaces, transcriptive mappings and *weak* descriptive mappings is that the compiler must make the callee handle remappings as a default case. But the callee has both less information and optimization opportunities [11].

These features improve expressiveness, but at the price of performance. Delaying to run time the array mapping handling of references means delaying the actual address calculations and reduces compile time optimizations which are mandatory to cache-based processors. Also compiling for an unknown mapping makes many communication optimizations impractical. Expensive and more complex techniques can be used to generate good code when lacking information: partial or full cloning of subroutines to be compiled with different assumptions, that requires a full interprocedural analysis and compilation [18]. Another technique is run time partial evaluation that dynamically generates an optimized code once enough information is available [7]. However even though there are overheads and the runtime is complex.

As HPF is expected to bring *high performance*, transcriptive and ambiguous mappings seem useless. They restrict the implementor choices and possible optimizations. Moreover no *real-life* application we have encountered so far require them to reach *high performance* levels.

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

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.

The remapping graph is a very small subgraph of the control flow graph. The vertices of the graph are the remapping statements whether explicit or added to model implicit remappings at call sites. There is a subroutine entry point vertex v_0 and an exit point v_e. An edge denotes a possible path in the control flow graph with the same array remapped at both vertices. The vertices are labeled with the remapped arrays. Each remapped array is associated *one* leaving copy and reaching copies at this vertex. Arrays are also associated a conservative use-information: Namely whether a given leaving copy may be not referenced (`N`), fully redefined before any use (`D`), only read (`R`) or maybe modified (`W`).

$$\mathtt{A}\,\{1,3\} \xrightarrow{\mathtt{R}} 2$$

Figure 9: Label representation

Figure 9 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 vertex is a remapping for array `A`. It may be reached with copies $\mathtt{A}_1$ and $\mathtt{A}_3$ and must be left with copy $\mathtt{A}_2$. As this copy will only be read, the compiler and runtime can decide to keep the reaching copy values which are live. A shorthand is used in some figures when several arrays share the same reaching and leaving mappings: All concerned arrays are specified as a prefix, and the use information over the arrow is specified for each array, respectively.

This provides a precise liveness 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. The remapping graph definition is more formally presented in Appendix A.

3.2 Construction

The remapping graph described above holds all the remapping and liveness information. The next issue is to build this graph. The construction algorithm 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.

Subroutine argument mappings are handled as local remappings by the caller. Implicit remappings are translated into explicit ones at call site in the caller. The actual array argument is copied if needed into a copy mapped as the corresponding dummy argument before the call, and may be copied back on return. The intent attribute (`in`, `out` or `inout`) provides information about the effects of the call onto the array and will be used to determine live copies. Within the subroutine compilation, three added vertices (call v_c, entry v_0 and exit v_e) model the initial and final mappings for the dummy arguments and local variables. Dummy arguments and local arrays are associated their initial mapping on exit from vertex v_0. v_c and v_e allow to attach dummy arguments the use information derived from the `intent` attribute to model imported and exported values.

Then the construction starts by propagating the initial mapping copy of the array from the entry point of the subroutine. The $\mathcal{G}_R$ 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. This construction can be described as a set of data-flow problems detailed in Appendix B. Mappings are propagated from the entry point and updated at remapping statements. This can be decomposed into two dataflow problems, one for alignments and one for distributions. However our implementation performs both propagation concurently, focussing directly on array mappings. The propagation tags array references with their associated mappings and performs some transformations to handle subroutine calls. Second, the use information is propagated backwards from references to remapping statements. Finally the contracted graph is defined by propagating remapping statements over the control graph.

3.3 Example

Let us focus on the routine in Figure 10. It 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. The sequential loop structure with two remappings is typical of ADI.

Figure 11 shows the resulting remapping graph. The liveness information is 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 [11] 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 run time conditions. These optimizations on $\mathcal{G}_R$ are expressed as standard data flow problems [14, 13, 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 dataflow problem. It is detailed in appendix C. First useless remappings are removed (unused leaving mappings). Second reaching mappings are computed again from remaining leaving mappings. This optimization is shown correct: All remapping that are useless are removed, and all those that may be useful are kept. Thus it is optimal, provided that remappings remain in place.

Figure 12 displays the remapping graph of our example after optimization. From the graph it results 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.

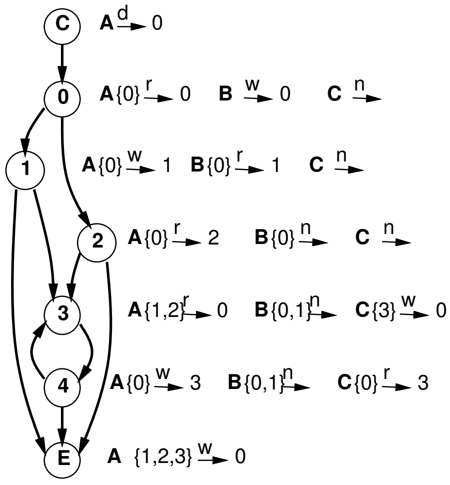

Figure 12: Example after optimization

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 13 and its corresponding remapping graph in Figure 14. 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: dead copies will be deleted (or mark as dead) at the remapping statements.

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. This is detailed in Appendix D.

4.3 Other optimizations

Further optimization can be thought of, as discussed in [11]. Array kill analysis, for instance based on array regions [8, 9], tells whether the values of an array are dead at a given point in the program. This semantical analysis can be used to

```
  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 10: Code example

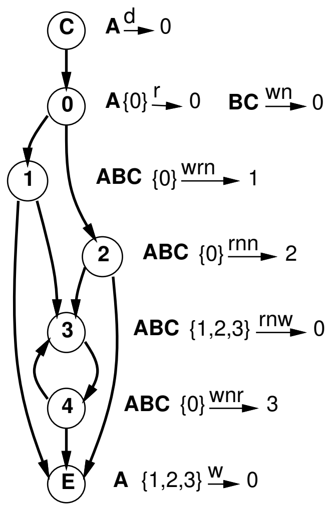

Figure 11: Remapping graph for Figure 10

```
!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 13: Flow dependent live copy

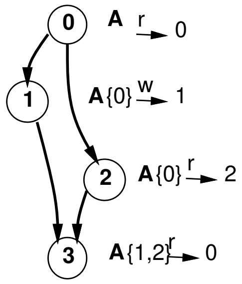

Figure 14: Corresponding $\mathcal{G}_R$

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 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 creates a remapping vertex tagged `D`.

Remappings can be moved around in the control flow graph, especially out of loops. From the code in Figure 15 we suggest to move the remappings as shown in Figure 16. This differs from [11]: the initial remapping is not moved out of the loop, because if `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 describing array versions reaching and leaving remapping vertices must be embedded into the program through actual copies in order to fulfill the requirements. Some optimizations described in the previous sections rely on the runtime to be performed.

```
      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 17: Subroutine calls

5.1 Runtime status

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) and the live copies.

The current status of an array can be kept in a descriptor holding the version number. By testing this status, the runtime is able to notice which version of an array reaches a remapping statement, what may be flow-dependent. This

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

Figure 15: Loop invariant remappings

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

Figure 16: Optimized version

```
! 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 18: Mapping restored

descriptor enables the handling of programs with ambiguous mappings provided that no actual reference to such an array is performed before a remapping. In order to test whether a version of a given array is live at a point, a boolean information to be attached to each array version. It will be updated at each remapping vertex, depending of the latter use of the copies from this vertex.

If interpreted strongly, Constraint 1 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 17. 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 18. 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 liveness information management from the remapping graph is outlined in Figure 19. Copy allocation and deallocation are inserted in the control flow graph to perform the required remappings, using the sets computed at the $\mathcal{G}_R$ optimization phase.

The first loop inserts the runtime management initialization at the entry point. All copies are denoted as not live. No copy receives an *a priori* instantiation. The rationale for doing so is to delay this instantiation to the actual use of the array, that may occur with a different mapping or never, as Array `C` in Figure 10. 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. 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 local arrays is inserted at the exit vertex. Figure 20 shows a generated copy code for

for A $\in S(v_0)$
  append to v_0 "status(A)=$\perp$"
  for $a \in C(\mathtt{A})$
    append to v_0 "live($\mathtt{A}_a$)=false"
  end for
end for
for $v \in \mathcal{V}(\mathcal{G}_R) - \{v_c\}$
  for A $\in S(v)$
    if ($L_{\mathtt{A}}(v) \neq \perp$) then
      append to v "if (status(A)$\neq L_{\mathtt{A}}(v)$) then"
      append to v "allocate $\mathtt{A}_{L_{\mathtt{A}}(v)}$ if needed"
      append to v "if (not live($\mathtt{A}_{L_{\mathtt{A}}(v)}$)) then"
      if ($U_{\mathtt{A}}(v) \neq \mathtt{D}$) then
        for $a \in R_{\mathtt{A}}(v) - \{L_{\mathtt{A}}(v)\}$
          append to v "if (status(A)=a) $\mathtt{A}_{L_{\mathtt{A}}(v)}$=$\mathtt{A}_a$"
        end for
      end if
      append to v "live($\mathtt{A}_{L_{\mathtt{A}}(v)}$)=true"
      append to v "endif"
      append to v "status(A)=$L_{\mathtt{A}}(v)$"
      append to v "endif"
    end if
    for $a \in C(\mathtt{A}) - M_{\mathtt{A}}(v)$
      append to v "if (live($\mathtt{A}_a$)) then"
      append to v " free $\mathtt{A}_a$ if needed"
      append to v " live($\mathtt{A}_a$)=false"
      append to v "endif"
    end for
  end for
end for
for all A
  for $a \in C(\mathtt{A})$
    append to v_e "if (live($\mathtt{A}_a$) and needed) free $\mathtt{A}_a$"
  end for
end for all

Figure 19: Copy code generation algorithm

```
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 20: Code for Figure 9

the remapping vertex in Figure 9.

It must be noted that dead arrays (D) do not require any actual array copy, thus none is generated, avoiding 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. Finally, care must be taken not to free the array dummy argument copy which belongs to the caller.

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, 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

In this paper, we have shown a pratical approach to compile HPF dynamic mappings. It consists of substituting dynamic arrays by static ones, and of inserting simple array copies between these arrays when necessary. Implicit remappings at call site are translated into explicit ones in the caller. We have discussed the language restrictions needed to apply this scheme, and argued that no high performance application should miss the restricted features. We have also presented optimizations enabled by our technique, to remove useless remappings and to detect live copies that can be reused without communication. Finally runtime implications have been discussed.

Most of the techniques described in this paper are implemented in our prototype HPF compiler [3]. It is available from `http://www.cri.ensmp.fr/pips/hpfc.html`. 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.

Acknowledgment

I am thankful to Corinne Ancourt, Béatrice Creusillet, François Irigoin, Pierre Jouvelot and to the anonymous referees 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] Jean-Yves Berthou and Laurent Colombet. Experiences in Data Parallel Programming on Cray MPP machines. First High Performance Fortran (HPF) Users Group Conference, Santa Fe, NM, USA, February 1997.

[3] Fabien Coelho. *Contributions to High Performance Fortran Compilation.* PhD thesis, École des mines de Paris, October 1996.

[4] Fabien Coelho. Discussing HPF Design Issues. In *Euro-Par'96, Lyon, France*, pages I.571–I.578, August 1996. LNCS 1123. Also report EMP CRI A-284, Feb. 1996.

[5] Fabien Coelho and Corinne Ancourt. Optimal Compilation of HPF Remappings. *Jounal of Parallel and Distributed Computing*, 38(2):229–236, November 1996. Also TR EMP CRI A-277 (October 1995).

[6] Fabien Coelho and Henry Zongaro. ASSUME directive proposal. TR A 287, CRI, École des mines de Paris, April 1996.

[7] Charles Consel and François Noël. A General Approach for Run-Time Specialization and its Application to C. In *Symposium on Principles of Programming Language*, pages 145–156, January 1996.

[8] Béatrice Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des mines de Paris, December 1996.

[9] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996.

[10] 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.

[11] 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.

[12] HPF Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1996. version 2.0.

[13] 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.

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

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

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

[17] 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.

[18] Daniel J. Palermo, Eugene W. Hodges IV, and Prithviraj Banerjee. Interprocedural Array Redistribution Data-Flow Analysis. In *Language and Compilers for Parallel Computing*, pages aa.1–aa.15, August 1996. San José, CA.

[19] Loïc Prylli and Bernard Tourancheau. Efficient Block Cyclic Data Redistribution. In *Euro-Par'96, Lyon, France*, pages I.155–I.164, August 1996. LNCS 1123. Also INRIA RR 2766.

```
!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 21: Several leaving mappings

A Remapping Graph Definition

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 $\mathrm{pred}(v)$.

vertices $\mathcal{V}(\mathcal{G}_R)$: the vertices are the remapping statements. They can be explicit (`realign`, `redistribute`) or added in place of implicit remappings at call sites. There is a subroutine entry point vertex v_0 and an exit point v_e.

edges $\mathcal{E}(\mathcal{G}_R)$: each edge denotes a possible path in the control flow graph with the same array remapped at both vertices and not remapped in between.

labels: in the remapping graph, each vertex v is associated $S(v)$, the set of remapped arrays.

For each array $\mathtt{A} \in S(v)$ we have some associated information (depicted in Figure 9):

$L_{\mathtt{A}}(v)$: *The* (or none, noted $\perp$) leaving array copy, *i.e.* the copy which must be referenced after the remapping; note that HPF allows several leaving mappings as depicted in Figure 21: 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 this presentation.

$R_{\mathtt{A}}(v)$: the set of reaching copies for the Array `A` at Vertex v.

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

$U_{\mathtt{A}}(v)$: describes how the leaving copy might be used afterwards. It may be never 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.

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

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

B Remapping Graph Construction

Here is 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 $\mathcal{G}_R$:

$\textsc{Reaching}(v)$: 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.

$\textsc{Leaving}(v)$: the set of arrays and associated mappings leaving vertex v; one leaving mapping per array is assumed for simplifying the presentation.

$\textsc{Remapped}(v)$: 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).

$\textsc{EffectsOf}(v)$: the proper effect on distributed variables of vertex v, *i.e.* these variables and whether they are never referenced, fully redefined, partially defined or used. This basic information is assumed to be available.

$\textsc{EffectsAfter}(v)$: the distributed variables and associated effects that may be encountered after v and before any remapping of these variables.

$\textsc{EffectsFrom}(v)$: just the same, but including also the effects of v.

$\textsc{RemappedAfter}(v)$: the distributed variables and associated remapping vertices that may be encountered directly (without intermediate remapping) after v.

$\textsc{RemappedFrom}(v)$: just the same, but including also v.

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

$\mathtt{A_j} = \textsc{impact}(\mathtt{A_i}, v)$: the resulting mapping of `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 `A` or redistributions of the template $\mathtt{A_i}$ is aligned with may give a new mapping. The impact of a call is null.

$\textsc{array}(\mathtt{A_i})=\mathtt{A}$: the function returns the array from one of its copies.

operator $-$: means *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.

intent	$U_{\mathtt{A}}(v_c)$	$U_{\mathtt{A}}(v_e)$
in	D	N
inout	D	W
out	N	W

Figure 22: Array argument use

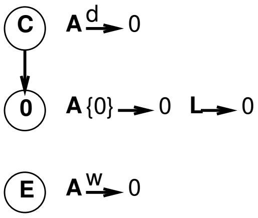

Figure 23: Initial $\mathcal{G}_R$

Input to the construction algorithm

- control flow graph $\mathcal{G}_C$ with entry v_0 and exit v_e vertices
- the set of remapping vertices V_R, which includes Vertex v_0 and Vertex v_e.
- the proper effects of vertices on distributed variables $\textsc{EffectsOf}(v)$ (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 $\mathcal{G}_C$.

Reaching and Leaving mappings

They 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 $\textsc{Leaving}(v_c)$ and $\textsc{Leaving}(v_e)$.
- update $\textsc{EffectsOf}(v_c)$ and $\textsc{EffectsOf}(v_e)$ as suggested in Figure 22: 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 is assumed to comply to the intended semantics.

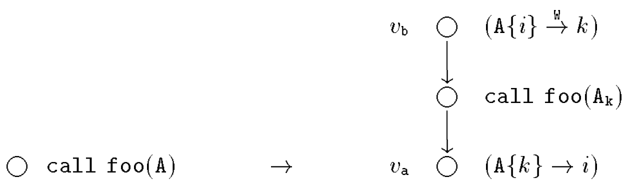

Figure 24: Call with a prescriptive `inout`-intended argument

- add all local distributed variables and their associated initial mapping to $\textsc{Leaving}(v_0)$.

Figure 23 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 statement impact 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

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

$$\textsc{Remapped}(v) = \bigcup_{\frac{1}{0}pt] m \in \textsc{Reaching}(v) \, m \neq \textsc{impact}(m,v)} \textsc{array}(m)$$

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

- calls with distributed arguments are managed as shown in Figure 24:
  $\mathrm{pred}(v_b) = \mathrm{pred}(v), \mathrm{succ}(v_b) = \{v\}, \mathrm{pred}(v_a) = \{v\}, \mathrm{succ}(v_a) = \mathrm{succ}(v), \mathrm{pred}(v) = \{v_b\}, \mathrm{succ}(v) = \{v_a\}$
  $\textsc{Remapped}(v_b) = \{\mathtt{A}\}$
- 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 considered array copy.

This phase is based on 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 associated to a subroutine argument the corresponding effect is described in Figure 25.

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

Figure 25: 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.

$$\textsc{EffectsAfter}(v) = \bigcup_{v' \in \mathrm{succ}(v)} \textsc{EffectsFrom}(v')$$

- 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 direct edges that will be used to propagate remapping information and optimizations quickly. This phase propagates for once remapping statements (array and vertex couples) so that each remapping statement will know its 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

Discussion

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 described 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 $\mathcal{O}(nsm^2p^2)$.

C 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. 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.

Remove useless remappings

Done simply by deleting the leaving mapping of such arrays.

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

Recompute reaching mappings

initialization: use 1-step reaching mappings

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

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

propagation: optimizing function

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

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 $\mathcal{O}(1)$ basic set element operations (put, get and membership). 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.

Correctness and Optimality

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):*

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

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.

D Dynamic live copies

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 in Figure 13, 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

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

Maybe useful copies are propagated backwards while the array is not modified (neither W nor D).