appeared in the 9th Workshop on Languages and Compilers for Parallel Computing, San Jose, CA, Aug. 8-10, 1996

Interprocedural Array Redistribution Data-Flow Analysis

Daniel J. Palermo,[1] Eugene W. Hodges IV,[2] and Prithviraj Banerjee[3]

[1] Hewlett-Packard Company, Convex Division, Richardson, TX 75083 (palermo@rsn.hp.com)
[2] SAS Institute Inc., Cary, NC 27513 (sasehx@unx.sas.com)
[3] University of Illinois at Urbana-Champaign, Urbana, IL 61801 (banerjee@crhc.uiuc.edu)

Abstract. In High Performance Fortran (HPF), array redistribution can be described explicitly using directives (`REDISTRIBUTE` or `REALIGN`) which specify where new distributions become active or implicitly by calling functions which require different data distributions than the calling function. In order to actually compile an HPF program into an efficient form, however, both the redistribution operations as well as the possible distributions for the individual blocks of code must be known at compile-time. In this paper, we present an interprocedural data-flow framework which takes into account both explicit and implicit redistribution to automatically: (1) determine which distributions hold over specific sections of a program, (2) optimize both the inter- and intraprocedural transitions between dynamic distributions while still maintaining the original semantics of the HPF program, (3) determine when the distribution pattern specified by an HPF program causes a given array to be assigned multiple distributions due to different redistribution operations on multiple paths within a function or as a result of parameter aliasing (resulting in a non-conforming HPF program), as well as (4) convert (well behaved) dynamic HPF programs into equivalent static forms through a process we refer to as static distribution assignment (SDA) which can be used to extend the capabilities of existing subset HPF compilers that support static data distributions. As the approach presented in this paper has already been implemented as part of the PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers) project at the University of Illinois, examples will also be presented to demonstrate several applications of this framework.

1 Introduction

For complex programs, static data distributions may be insufficient to obtain acceptable performance on distributed-memory multicomputers. By allowing the data distribution to dynamically change over the course of a program's execution this problem can be alleviated by matching the data distribution more closely to the different computations performed throughout the program. Such dynamic partitionings can yield higher performance than a static partitioning when the redistribution is more efficient than the communication pattern required by the statically partitioned computation.

In High Performance Fortran (HPF), dynamic distributions can be described explicitly using executable redistribution directives (`REDISTRIBUTE` or `REALIGN`, which spec-

This research, performed at the University of Illinois, was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613, in part by an Office of Naval Research Graduate Fellowship, and in part by the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research office.

ify where new distributions become active) or implicitly by calling functions[4] (which require different data distributions than the calling function). To actually compile an HPF program into an efficient form, however, both the redistribution operations as well as the possible distributions for the individual blocks of code must be known. Since the HPF `REDISTRIBUTE` and `REALIGN` directives only specify redistribution information, both the intra- and interprocedural control flow through a program must be examined to decide exactly which redistribution operations were last performed in order to determine which distributions are active at any given point in the program.

For function calls, the distribution of the actual arguments which appear in a call to a function will not necessarily match the distribution of the dummy arguments within the function. In HPF, several different forms of distribution directives for dummy arguments are provided [11]. The distribution directive for a dummy argument may be:

prescriptive - The distribution *prescribes* the mapping of the dummy argument. Passing an actual argument with a different distribution will require redistributing the argument.

transcriptive - The distribution of the dummy argument is inherited or *transcribed* from the actual argument.

descriptive - The distribution *describes* the mapping of the dummy argument with the claim that no redistribution will take place. If an interface is present in the calling function, then this becomes a prescriptive directive.

Both prescriptive as well as descriptive arguments (with a provided interface) can have the same effect as explicit `REDISTRIBUTE` or `REALIGN` directives whereas transcriptive arguments provide a mechanism for writing functions which can accept any type of distribution. As purely descriptive directives (without an interface) do not require any additional support from an HPF compiler, they will not be discussed further in this paper.

The basic motivation for the framework presented in this paper is best summarized in the following excerpt from the HPF language specification [10]:

> *An overriding principle is that any mapping or remapping of arguments is not visible to the caller. This is true whether such remapping is implicit (in order to conform to prescriptive directives, which may themselves be explicit or implicit) or explicit (specified by `REALIGN` or `REDISTRIBUTE` directives). When the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. It is not possible for a subprogram to change the mapping of any object in a manner visible to its caller, not even by means of `REALIGN` and `REDISTRIBUTE`.*

Interprocedural array data-flow analysis can be used to provide the correct semantics in the presence of implicit and explicit redistributions (or remappings[5]) and generate

[4] For the purposes of simplifying this discussion, the word *function* will be used throughout this paper to refer to either a Fortran program or subprogram (function or subroutine) and should be considered to be interchangeable with all of these terms.

[5] In the HPF standard [10], the word *remapping* is used to refer to *redistribution* in order to avoid associating it with only the HPF `REDISTRIBUTE` directive. For this paper, the word *redistribution* will be used equivalently to refer to remapping due to either the HPF `REDISTRIBUTE` or `REALIGN` directive.

efficient code. Such analysis avoids using either a simple copy-in/copy-out strategy or a complete redistribution of all arguments upon every entry and exit of a function. As will be shown, it is possible to optimize such interprocedural redistribution simultaneously with intraprocedural redistribution operations within the framework.

The remainder of this paper is organized as follows: related work in array redistribution and interprocedural analysis is discussed in Section 2; Section 3 presents an overview of the basic approach of the framework and the representations used in its development; the techniques for performing interprocedural array redistribution analysis are presented in Section 4; an example of the application of these techniques is given in Section 5; and conclusions are presented in Section 6.

2 Related Work

Interprocedural analysis is a useful program analysis technique that has been applied to many different applications [1]. In the area of parallel processing, it has been widely used for dependence analysis and program parallelization in the presence of function calls [3, 8, 9, 16], detecting references to stale data for improving the performance of cache coherent architectures [4], as well as for compiling static Fortran D programs [7].

The work by Hall, Hiranandani, Kennedy, and Tseng [7] defined the term *reaching decompositions* for the Fortran D decompositions (or distributions, in HPF terminology) which reach a function call site. Their work describes extensions to the Fortran D compilation strategy [17] using the reaching decompositions for a given call site to compile Fortran D programs that contain function calls as well as to optimize the resulting implicit redistribution. As presented, their techniques addressed computing and optimizing (redundant or loop invariant) implicit redistribution operations due to changes in distribution at function boundaries, but do not address many the situations which arise in HPF.

Work by Coelho and Ancourt [5] also describes an optimization for removing useless remappings specified by a programmer through explicit realign and redistribute operations. In comparison to the work in the Fortran D project, they are also concerned only with determining which distributions are generated from a set of redistributions, but instead focus only on explicit redistribution. They define a new representation called a redistribution graph in which nodes represent redistribution operations and edges represent the statements executed between redistribution operations. This representation, although correct in its formulation, does not seem to fit well with any existing analysis already performed by optimizing compilers and also requires first summarizing all variables used or defined along every possible path between successive redistribution operations in order to optimize redistribution. Even though their approach currently only performs their analysis within a single function, they do suggest the possibility of an extension to their techniques which would allow them to also handle implicit remapping operations at function calls but do not describe an approach.

The definition of reaching distributions, however, is still a useful concept. We extend this definition to also include distributions which reach any point within a function in order to encompass both implicit and explicit redistributions thereby forming the basis of the work presented in this paper. In addition to determining those distributions generated from a set of redistribution operations, this extended definition allows us to address a number of other applications in a unified framework.

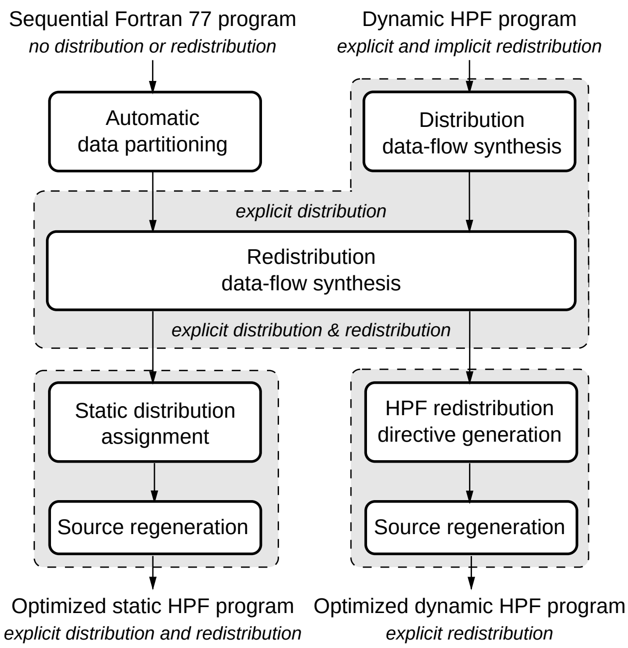


Fig. 1. Overview of the array redistribution data-flow analysis framework (shaded areas indicate components described in this paper)

3 Data Redistribution Analysis

An overall view of the array redistribution data-flow analysis framework we have developed is shown in Figure 1. In addition to analyzing HPF programs, the framework was also developed as a back-end to an automatic data partitioning system [13, 12] also developed as part of the PARADIGM compilation system [2] in order to provide a mechanism to generate fully explicit dynamic HPF programs. The shaded areas of the Figure indicate the components of the system which will be described in this paper.

The intermediate form of a program within the framework is one in which the distribution of every array at every point in the program as well as the redistribution required to move from one point to the next are explicitly known. The different paths through the framework involve passes which process the available distribution information in order to obtain the missing information required to move from one representation to another.

In HPF, dynamic distributions are described by specifying the transitions between different distributions (through explicit redistribution directives or implicit redistribution at function boundaries). Dynamic HPF programs (in which every use or definition of an array only has one reaching redistribution) can be converted to static versions (with explicit redistribution and distribution) through a process we call static distribution assignment (SDA) which will be explained later in this paper. With this framework, it is also possible to convert arbitrary HPF programs into an optimized HPF program containing only explicit redistribution directives and descriptive function arguments. The data partitioner, on the other hand, explicitly assigns different distributions to individual blocks of code serving as an automated mechanism for converting sequential Fortran programs into efficient HPF programs. In this case, the framework can be used to synthesize explicit interprocedural redistribution operations in order to preserve the meaning of what the data partitioner intended in the presence of HPF semantics.

The core of this analysis is built upon two separate interprocedural data-flow problems which perform distribution synthesis (Section 4.1), and redistribution synthesis (Section 4.2). These two data-flow problems are both based upon the problem of determining both the inter- and intraprocedural reaching distributions for a program. Before giving further details of how all of these transformations are accomplished through the use of these two data-flow problems, we will first describe the idea of reaching distributions and the basic representations we use to perform this analysis.

3.1 Computing reaching distributions

The problem of determining which distributions reach any given point taking into account control flow in the program is very similar to the computation of reaching definitions (a forward data-flow problem):

> *A definition d is said to reach a point p if there is a path in the control flow graph (CFG) from the point immediately following d to p, such that d is not* killed *along that path by another definition* [1].

In classic compilation theory a control flow graph consists of nodes (basic blocks) representing uninterrupted sequences of statements and edges representing the flow of control between basic blocks. For determining reaching distributions, an additional restriction must be added to this definition. Not only should each block B be viewed as a sequence of statements with flow only entering at the beginning and leaving at the end, but the data distribution for the arrays defined or used within the block is also not allowed to change. In comparison to the original definition of a basic block, this imposes tighter restrictions on the extents of a block. Using this definition of a block in place of a basic block results in what we refer to as the distribution flow graph (DFG). This representation differs from [5] as redistribution operations now merely augment the definition of basic block boundaries as opposed to forming the nodes of the graph.

Using this view of a block in a DFG and by viewing array distributions as definitions, the same data-flow framework used for computing reaching definitions [1] can now be used to obtain the reaching distributions by defining the following sets for each block in a function:

- DIST(B) - set of distributions present when executing block B
- REDIST(B) - set of redistributions performed upon entering block B
- GEN(B) - set of distributions generated by executing block B
- KILL(B) - set of distributions killed by executing block B
- IN(B) - set of distributions that exist upon entering block B
- OUT(B) - set of distributions that exist upon leaving block B
- DEF(B), USE(B) - set of variables defined or used in block B

It is important to note that GEN and KILL are specified as the distributions generated or killed by *executing* block B as opposed to entering (redistribution at the head of the block) or exiting (redistribution at the tail of the block) in order to allow both forms of redistribution. GEN and KILL are initialized by DIST or REDIST (depending on the current applications as described in Section 4) and may be used to keep track of redistributions that occur on entry (e.g., HPF redistribute directives or functions with prescriptive distributions) or exit (e.g., calls to functions which internally change a distribution before returning). To perform interprocedural analysis, the function itself also has IN and OUT sets, which contain the distributions present upon entry and summarize the distributions for all possible exits.

Once the sets have been defined, the following data-flow equations are iteratively computed for each block until the solution $\mathrm{OUT}(B)$ converges for every block B (where $\mathrm{PRED}(B)$ are the nodes which immediately precede B in the flow of the program):

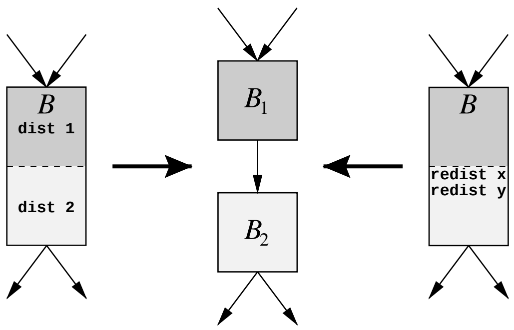
$$\mathrm{IN}(B) = \bigcup_{P \in \mathrm{PRED}(B)} \mathrm{OUT}(P) \tag{1}$$

$$\mathrm{OUT}(B) = \mathrm{GEN}(B) \bigcup (\mathrm{IN}(B) - \mathrm{KILL}(B)) \tag{2}$$

Since the confluence operator is a union, both IN and OUT never decrease in size and the algorithm will eventually halt. By processing the blocks in the flow graph in a depth-first order, the number of iterations performed will roughly correspond to the level of the deepest nested statement, which tends to be a fairly small number on real programs [1].

As can be seen from Eqs. (1) and (2), the DEF and USE sets are actually not used to compute reaching distributions, but will have other uses for optimizing redistribution which will be explained in more detail in Section 4.2).

3.2 Constructing the distribution flow graph



(a) Distribution split (b) Redistribution split

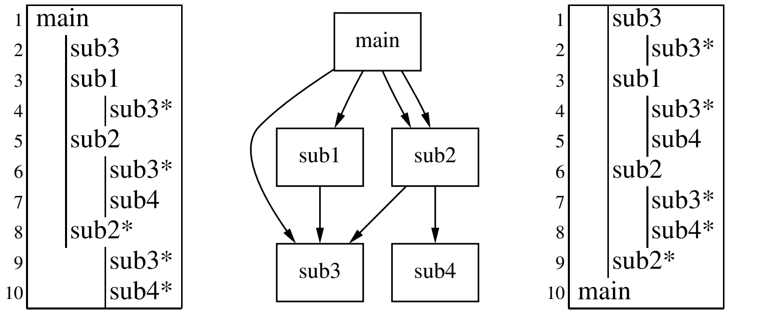
Fig. 2. Splitting CFG nodes to obtain DFG nodes

Since the definition of the DFG is based upon the CFG, the CFG can be easily transformed into a DFG by splitting basic blocks at points at which a distribution changes as shown in Figure 2. This can be due to an explicit change in distribution, as specified by the automatic data partitioner, or by an actual HPF redistribution directive. If the change in distribution is due to a sequence redistribution directives, the overall effect is assigned to the block in which they are contained; otherwise, a separate block is created whenever executable operations are interspersed between the directives.

4 Interprocedural Data Redistribution Analysis

Since the semantics of HPF require that all objects accessible to the caller after the call are distributed exactly as they were before the call, it is possible to first completely examine the context of a call before considering any distribution side effects due to the call. It may seem strange to say that there can be side effects when we just said that the semantics of HPF preclude it. To clarify this statement, such side effects *are* allowed to exist, but only to the extent that they are not *apparent* outside of the call. As long as the view specified by the programmer is maintained, the compiler is allowed do whatever it can to optimize both the inter- and intraprocedural redistributions so long as the resulting distributions used at any given point in the program are not changed.

Referring back to Figure 1 the core of this analysis is built upon two separate interprocedural data-flow problems which perform distribution synthesis and redistribution synthesis. These two data-flow problems are based upon the problem of determining both the inter- and intraprocedural reaching distributions for a program. An example call graph is shown in Figure 3 to help illustrate the flow of these two phases of the interprocedural analysis.



(a) Top-down (pre-order) for calling context

(b) Bottom-up (post-order) for side-effects

Fig. 3. Example call graph and depth-first traversal order

If distribution information is not present (i.e., HPF input), distribution synthesis is first performed in a top-down manner over a program's call graph to compute which distributions are present at every point within a given function. By establishing the distributions that are present at each call site, the input distributions are obtained for each of the functions which it calls as well as causes a function to be cloned if a new set of input distributions is used.[6] Redistribution synthesis is then applied in a bottom-up manner over the call graph to analyze where the distributions are actually used and synthesizing the redistribution required within a function.

Since this analysis is interested in the effects between an individual caller/callee pair, and not in summarizing the effects from all callers before examining a callee, it is not necessary to perform a topological traversal for the top-down and bottom-up passes over

[6] Because there is considerable state associated with the body of a function, it is much more efficient to record the cloned data as a separate context for the function and delay the actual cloning until later, a process which we refer to as *virtual cloning*.

the call graph. In this case, it is actually more intuitive to perform a depth-first pre-order traversal of the call graph (shown in Figure 3(a)) to fully analyze a given function before proceeding to analyze any of the functions it calls and to perform a dept-first post-order traversal (shown in Figure 3(b)) to fully analyze all called functions before analyzing the caller.

One other point to emphasize is that these interprocedural techniques can be much more efficient than analyzing a fully inlined version of the same program since it is possible to prune the traversal at the point a previous solution is found for a function in the same calling context. In Figure 3, asterisks indicate points at which a function is being examined after having already been examined previously. If the calling context is the same as the one used previously, the traversal can be pruned at this point reusing information recorded from the previous context. Depending on how much reuse occurs, this factor can greatly reduce the amount of time the compiler spends analyzing a program in comparison to a fully inlined approach.

The technique for performing distribution synthesis will be described in Section 4.1 while redistribution synthesis will be described in Section 4.2. The static HPF conversion, known as static distribution assignment (SDA), is covered in detail in Section 4.3. Since the dynamic HPF conversion only entails generating redistribution directives based on the contents of the REDIST sets, it will not be discussed further in this section. More detailed descriptions of the actual implementation of these techniques can be found in [12].

4.1 Distribution synthesis

When analyzing HPF programs, it is necessary to first perform distribution synthesis in order to determine which distributions are present at every point in a program. Since HPF semantics specify that any redistribution (implicit or explicit) due to a function call is not visible to the caller, each function can be examined independently of the functions it calls. Only the input distributions for a given function and the explicit redistribution it performs have to be considered to obtain the reaching distributions for a function.

Given an HPF program, nodes (or blocks) in its DFG are delimited by the redistribution operations which appear in the form of HPF `REDISTRIBUTE` or `REALIGN` directives. As shown in Figure 4, the redistribution operations assigned to a block B represent the redistribution that will be performed when entering the block on any input path (indicated by the set REDIST(B)) as opposed to specifying the redistribution performed for each incoming path (REDIST(B, B_1) or REDIST(B, B_2) in the figure).

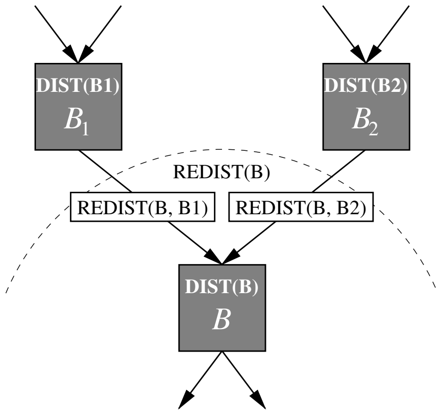


Fig. 4. Distribution synthesis (converting redistributions to distributions)

If the set GEN(B) is viewed as the distributions which are generated and KILL(B) as the distributions which are killed upon entering the block, this problem can now be cast directly into the reaching distribution data-flow framework by making the following assignments:

Data-flow initialization:

$$\begin{aligned} \text{REDIST}(B) &= \text{from directives} & \text{DIST}(B) &= \emptyset \\ \text{GEN}(B) &= \text{REDIST}(B) & \text{KILL}(B) &= \overline{\text{REDIST}_{var}(B)} \\ \text{OUT}(B) &= \text{REDIST}(B) & \text{IN}(B) &= \emptyset \end{aligned}$$

Data-flow solution:

$$\text{DIST(B)} = \text{OUT}(B)$$

According to the HPF standard, a `REALIGN` operation only affects the array being realigned while a `REDISTRIBUTE` operation should redistribute all arrays currently aligned to the given array being redistributed (in order to preserve any previous specified alignments). The current implementation only records redistribution information for the array immediately involved in a `REDISTRIBUTE` operation. This results in only redistributing the array involved in the directive and not all of the alignees of the target to which it is aligned. In the future, the implementation could be easily extended to support the full HPF interpretation of `REDISTRIBUTE` by simply recording the same redistribution information for all alignees for the target template of the array involved in the operation. Due to the properties of `REALIGN`, this will also require first determining which templates arrays are aligned to at every point in the program (using similar techniques).

4.2 Redistribution synthesis

After the distributions have been determined for each point in the program, the redistribution can be optimized. Instead of using either a simple copy-in/copy-out strategy or a complete redistribution of all arguments upon every entry and exit of a function, any implicit redistribution around function calls can be reduced to only that which is actually required to preserve HPF semantics. Any unnecessary redistribution operations (implicitly specified by HPF semantics or explicitly specified by a programmer) that result in distributions which would not otherwise be used before another redistribution operation occurs are completely removed in this pass.

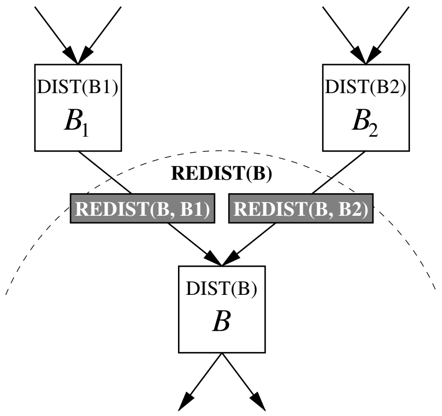


Fig. 5. Redistribution synthesis (converting distribution to redistribution)

Blocks are now delimited by changes in the distribution set. As shown in Figure 5, the set of reaching distributions previously computed for a block B represent the distributions which are in effect when executing that block (indicated by the set DIST$(()B)$). By first restricting the DIST(B) sets to the variables defined or used within block B, a redistribution operation will only be performed between two blocks if there is an intervening definition or use of that variable before the next change in distribution.

If the set GEN(B) is viewed as the distributions which are generated and KILL(B) as the distributions which are killed upon leaving block B, this problem can now be cast directly into the reaching distribution data-flow framework by making the following initializations:

data-flow initialization:

$$\begin{aligned} \text{REDIST}(B) &= \emptyset & \text{DIST}(B) &= \text{DIST}(B) \bigcap (\text{DEF}(B) \bigcup \text{USE}(B)) \\ \text{GEN}(B) &= \text{DIST}(B) & \text{KILL}(B) &= \overline{\text{DIST}_{var}(B)} \\ \text{IN}(B) &= \emptyset & \text{OUT}(B) &= \text{DIST}(B) \end{aligned}$$

data-flow solution:

$$\begin{aligned} \text{REDIST}(B,P) &= \text{DIST}_{var}(B) \mid (\text{OUT}_{var}(P) - \text{DIST}_{var}(B)) \neq \emptyset \ \ (\forall P \in \text{PRED}(B)) \\ \text{REDIST}(B) &= \bigcup_{P \in \text{PRED}(B)} \text{REDIST}(B,P) \\ &= \text{DIST}_{var}(B) \mid (\text{IN}_{var}(B) - \text{DIST}_{var}(B)) \neq \emptyset \end{aligned}$$

We define GEN(B) and KILL(B) as the distributions which are generated or killed upon leaving block B due to the fact that we have chosen to use a caller redistributes model. As will be seen later, this exposes many interprocedural optimization opportunities and is also necessary to support function calls which may require redistribution on both their entry and exit. Since REDIST(B) is determined from both the DIST and IN sets, DIST(B) represents the distributions needed for executing block B, while the GEN, KILL sets will be used to represent the exit distribution (which may or may not match DIST).

Optimizing invariant distributions Besides performing redistribution only when necessary, it is also desirable to only perform necessary redistribution as infrequently as possible. Semantically loop invariant distribution regions can be first grown before synthesizing the redistribution operations. All distributions that do not change within a nested statement (loop or if structures) are recorded on the parent statement (or header) for that structure. This has the effect of moving redistribution operations which result in the invariant distribution out of nested structures as far as possible (as was also possible in [7]). An example of this optimization will be illustrated later in Section 5.

As a side effect, loops which are considered to contain an invariant distribution no longer propagate previous distributions for the invariant arrays. Since redistribution is moved out of the loop, this means that for the (extremely rare) special case of a loop invariant distribution (which was not originally present outside of the loop) contained within a undetectable zero trip loop, only the invariant distribution from within the loop body is propagated even though the loop nest was never executed. As this is only due to the way invariant distributions are handled, the data-flow handles non-invariant distributions as expected for zero trip loops (an extra redistribution check may be generated after the loop execution).

Multiple active distributions Even though it is not specifically stated as such in the HPF standard, we will consider an HPF program in which every use or definition of an array has only one active distribution to be well-behaved. Since PARADIGM cannot currently compile programs which contain references with multiple active distributions, this property is currently detected by examining the reaching distribution sets for every node (limited by DEF/USE) within a function. A warning is issued if any set contains multiple distributions for a given variable stating that the program is not well-behaved. An example of this situation will be illustrated later in Section 5.

In the presence of function calls, it is also possible to access an array through two or more more paths when parameter aliasing is present. If there is an attempt to redistribute one of the aliased symbols, the different aliases now have different distributions even though they actually refer to the same array. This form of multiple active distributions is actually considered to be non-conforming in HPF [11] as it can result in consistency problems if the same array were allowed to occupy two different distributions. As it may be difficult for the programmers to make this determination, this can be automatically detected by determining if the reaching distribution set contains different distributions for any aliased arrays.[7]

4.3 Static Distribution Assignment (SDA)

To utilize the available memory on a given parallel machine as efficiently as possible, only the distributions that are active at any given point in the program should actually be allocated space. It is interesting to note that as long as a given array is distributed among the same total number of processors, the actual space required to store one section of the partitioned array is the same no matter how many array dimensions are distributed.[8] By using this observation, it is possible to statically allocate the minimum amount of memory by associating all possible distributions of a given array to the same area of memory.

Static Distribution Assignment (SDA) (inspired indirectly by the Static Single Assignment (SSA) form [6]) is a process we have developed in which the names of array variables are duplicated and renamed statically based on the active distributions represented in the corresponding DIST sets. As names are generated, they are assigned a static distribution (by which we mean this new name will not change distribution during the course of the program) corresponding to the currently active dynamic distribution for the original array. Redistribution now takes the form of moving data from a statically distributed source array to another statically distributed destination array (as opposed to rearranging the data within a single array).

To statically achieve the minimum amount of memory allocation required, all of the renamed duplicates of a given array are declared to be "equivalent." The `EQUIVALENCE` statement in Fortran 77 allows this to be performed at the source level in a somewhat similar manner as assigning two array pointers to the same allocated memory as is possible in C or Fortran 90, Redistribution directives are also now replaced with actual calls to a redistribution library.

Because the different static names for an array share the same memory, this implies that the communication operations used to implement the redistribution should read all

[7] The `param_alias` pass in Parafrase-2 [14] is first run to compute the alias sets for every function call.

[8] Taking into account distributions in which the number of processors allocated to a given array dimension does not evenly divide the size of the dimension, or degenerate distributions in which memory is not evenly distributed over all processors, can be equivalently said that there is an amount of memory which can store all possible distributions with very little excess.

of the source data before writing to the target. In the worst case, an entire copy of a partitioned array can be buffered at the destination processor before it is actually received and moved into the destination array. However, as soon as more than two different distributions are present for a given array, the `EQUIVALENCE` begins to pay off, even in the worst case, in comparison to separately allocating each different distribution. If the performance of buffered communication is insufficient for a given machine (due to the extra buffer copy), non-buffered communication could be used instead thereby precluding the use of `EQUIVALENCE` (unless some form of explicit buffering is performed by the redistribution library itself).

```
      REAL A(N, N)
!HPF$ DISTRIBUTE (CYCLIC, *) :: A
      ...
      A(i, j) = ...
      ...
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) :: A
      ...
      ... = A(i, j)
      ...
```

(a) Before SDA

```
      REAL A$0(N, N), A$1(N,N)
!HPF$ DISTRIBUTE (CYCLIC, *) :: A$0
!HPF$ DISTRIBUTE (BLOCK, BLOCK) :: A$1
      EQUIVALENCE (A$0, A$1)
      INTEGER A$cid
      A$cid = 0
      ...
      A$0(i, j) = ...
      ...
      CALL reconfig(A$1, 1, A$cid)
      ...
      ... = A$1(i, j)
      ...
```

(b) After SDA

Fig. 6. Example of static distribution assignment

In Figure 6, a small example is shown to illustrate this technique. In this example, a redistribution operation on `A` causes it to be referenced using two different distributions. A separate name is statically generated for each distribution of `A`, and the redistribution directive is replaced with a call to a run-time redistribution library [15]. The array accesses in the program can now be compiled by PARADIGM using techniques developed for programs which only contain static distributions [2] by simply ignoring the communication side effects of the redistribution call.

If more than one distribution is active for any given array reference, the program is considered to be *not well-behaved*, and the array involved can not be directly assigned a static distribution. In certain circumstances, however, it may be possible to perform code transformations to make an HPF program well-behaved. For instance, a loop that contained multiple active distributions on the entry to its body due only to a distribution from the loop back edge (caused by redistribution within the loop) that wasn't present on the loop entry would not be well-behaved. If the first iteration of that loop were peeled off, the entire loop body would now have a single active distribution for each variable and the initial redistribution into this state would be performed outside of the loop. This and other code transformations which help reduce the number of distributions reaching any given node will be the focus of further work in this area.

5 Examples

In Figure 7(a), a synthetic HPF program is presented which performs a number of different tests (described as comments in the input code) of the optimizations performed by the framework. In this program, one array, `x`, is redistributed both explicitly using HPF directives and implicitly through function calls using several different interfaces. Two of the functions, `func1` and `func2`, have prescriptive interfaces which may or may not require redistribution (depending on the current configuration of the input array). The first function differs from the second in that it also redistributes the array such that it returns with a different distribution than which it was called. The last function, `func3`, differs from the first two in that it has an (implicit) transcriptive interface. Calls to this function will cause it to inherit the current distribution of the actual parameters.

Several things can be noted when examining the optimized HPF shown in Figure 7(b). First of all, the necessary redistribution operations required to perform the implicit redistribution at the function call boundaries have been made explicit in the program. Here, the interprocedural analysis has completely removed any redundant redistribution by relaxing the HPF semantics allowing distributions caused by function side effects to exist so long as they do not affect the original meaning of the program. For the transcriptive function, `func3`, the framework generated two separate clones, `func3$0` and `func3$1`, corresponding to two different active distributions at a total of three different calling contexts.

Two warnings were also generated by the compiler, inserted by hand as comments in Figure 7(b), indicating that there were (semantically) multiple reaching distributions to two uses of `x` in the program. The first use actually does have two reaching distributions due to a conditional with redistribution performed on only one path. The second use, however, occurs after a call to a prescriptive function, `func1`, which implicitly redistributes the array to conform to its interface. Because a redistribution operation can only generate a single distribution, `x` can actually have only one reaching distribution, but semantically it still has two – hence the second warning.

As there are several other optimizations performed on this example, which have been described previously, we will not describe them in more detail here, but the reader is directed to the comment descriptions in the code for further information.

6 Conclusions

In this paper we have presented an interprocedural data-flow technique that can be used to transform redistribution to distributions or distributions to redistribution as well as optimize redistribution while always maintaining the semantics of the original program. Aliasing information is also used to ensure that the resulting distributions specified by the HPF program yield identical distributions for all aliased arrays. Other than for the support to physically perform redistribution in the middle of a statement (required for a return from a function call with different distributions) and support for array reshaping (which can occur through function parameters), these techniques have been fully implemented in the PARADIGM compiler [12].

Further work will also be focused on the application of code transformations (such as loop peeling) which will help reduce the number of reaching distributions to uses of variables with multiple active distributions. When it is not possible to completely eliminate such regions, code replication approaches, similar to the selective function cloning techniques used in this paper, will also be examined to allow an HPF compiler to efficiently handle such code regions (which are currently not supported in PARADIGM due to the difficulty in handling such situations).

```
      PROGRAM test
      INTEGER x(10,10)
c     *** For tests involving statement padding
      INTEGER a
!HPF$ PROCESSORS :: square(2,2)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, BLOCK) :: x
c     *** Use of initial distribution
      x(1,1) = 1
c     *** Testing loop invariant redistribution
      DO i = 1,10
         DO j = 1,10
            a = 0
!HPF$       REDISTRIBUTE (BLOCK, CYCLIC) :: x
            x(i,j) = 1
            a = 0
         ENDDO
      ENDDO
      a = 0
c     *** Testing unnecessary redistribution
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) :: x
      if (x(i,j) .gt. 1) then
c        *** Testing redistribution in a conditional
!HPF$    REDISTRIBUTE (BLOCK, BLOCK) :: x
         x(i,j) = 2
         call func3(x,n)
      else
         x(i,j) = 3
      endif
c     *** Uses with multiple reaching distributions
      x(1,1) = 2
      call func1(x,n)
      DO i = 1,10
         DO j = 1,10
            x(j,i) = 2
         ENDDO
      ENDDO
!HPF$ REDISTRIBUTE (CYCLIC(3), CYCLIC) :: x
c     *** Testing chaining of function arguments
      call func1(x,n)
      call func2(x,n)
      call func1(x,n)
c     *** Testing loop invariant due to return
      DO i = 1,10
         DO j = 1,10
c           *** Test cloning for transcriptive functions
            call func3(x,n)
         ENDDO
      ENDDO
      a = 1
c     *** Testing unused distribution
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) :: x
      a = 0
c     *** Testing "semantically killed" distribution
!HPF$ REDISTRIBUTE (CYCLIC(3), CYCLIC) :: x
      call func3(x,n)
      END

      integer function func1(a,n)
c     *** Prescriptive function with different return
      integer n, a(n, n)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,1) = 1
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,2) = 1
!HPF$ REDISTRIBUTE (CYCLIC, CYCLIC) :: a
      a(1,3) = 1
      end

      integer function func2(y,n)
c     *** Prescriptive function with identical return
      integer n, y(n,n)
!HPF$ DYNAMIC, DISTRIBUTE (CYCLIC, CYCLIC) :: y
      y(1,1) = 2
      end

      integer function func3(z,n)
c     *** (implicitly) Transcriptive function
      integer n, z(n,n)
      z(1,1) = 3
      end
```

```
      PROGRAM test
      INTEGER x(10,10)
      INTEGER a, n
!HPF$ PROCESSORS :: square(2,2)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, BLOCK) :: x
      x(1,1) = 1
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      DO i = 1,10
        DO j = 1,10
          a = 0
          x(i,j) = 1
          a = 0
        END DO
      END DO
      a = 0
      IF (x(i,j) .GT. 1) THEN
!HPF$ REDISTRIBUTE (BLOCK, BLOCK) ONTO square :: x
        x(i,j) = 2
        CALL func3$0(x,n)
      ELSE
        x(i,j) = 3
      END IF
c     *** WARNING: too many dists (2) for x
      x(1,1) = 2
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
      DO i = 1,10
        DO j = 1,10
c         *** WARNING: too many dists (2) for x
          x(j,i) = 2
        END DO
      END DO
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
      CALL func2(x,n)
!HPF$ REDISTRIBUTE (BLOCK, CYCLIC) ONTO square :: x
      CALL func1(x,n)
!HPF$ REDISTRIBUTE (CYCLIC(3), CYCLIC) ONTO square :: x
      DO i = 1,10
        DO j = 1,10
          CALL func3$1(x,n)
        END DO
      END DO
      a = 1
      a = 0
      CALL func3$1(x,n)
      END

      INTEGER FUNCTION func1(a,n)
      INTEGER n, a(n,n)
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK, CYCLIC) :: a
      a(1,1) = 1
      a(1,2) = 1
!HPF$ REDISTRIBUTE (CYCLIC, CYCLIC) ONTO square :: a
      a(1,3) = 1
      END

      INTEGER FUNCTION func2(y,n)
      INTEGER n, y(n,n)
!HPF$ DYNAMIC, DISTRIBUTE(CYCLIC,CYCLIC) ONTO square :: y
      y(1,1) = 2
      END

      INTEGER FUNCTION func3$1(n,z)
      INTEGER n, z(n,n)
!HPF$ DISTRIBUTE(CYCLIC(3),CYCLIC) ONTO square :: z
      z(1,1) = 3
      END

      INTEGER FUNCTION func3$0(n,z)
      INTEGER n, z(n,n)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO square :: z
      z(1,1) = 3
      END
```

(a) Before optimization   (b) After optimization

Fig. 7. Synthetic example for interprocedural redistribution optimization

(The HPF output, generated by PARADIGM, has been slightly simplified by removing unnecessary alignment directives from the figure to improve its clarity.)

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publ., Reading, MA, 1986.
2. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, Oct. 1995.
3. M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. In *Proc. of the ACM SIGPLAN Symp. on Compiler Construction*, pages 162–175, Palo Alto, CA, June 1986.
4. L. Choi and P.-C. Yew. Interprocedural Array Data-Flow Analysis for Cache Coherence. In *Proc. of the 8th Work. on Langs. and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 81–95, Columbus, OH, Aug. 1995. Springer-Verlag. 1996.
5. F. Coelho and C. Ancourt. Optimal Compilation of HPF Remappings (Extended Abstract). Tech. Report CRI A-277, Centre de Recherche en Informatique, École des mines de Paris, Fontainebleau, France, Nov. 1995.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Prog. Langs. and Sys.*, 13(4):451–490, Oct. 1991.
7. M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines. In *Proc. of Supercomputing '92*, pages 522–534, Minneapolis, MN, Nov. 1992.
8. M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. Interprocedural Analysis for Parallelization. In *Proc. of the 8th Work. on Langs. and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 61–80, Columbus, OH, Aug. 1995. Springer-Verlag. 1996.
9. P. Havlak and K. Kennedy. Experience with Interprocedural Analysis of Array Side Effects. In *Proc. of Supercomputing '90*, pages 952–961, New York, NY, Nov. 1990.
10. High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.1. Tech. report, Center for Research on Parallel Computation, Rice Univ., Houston, TX, Nov. 1994.
11. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
12. D. J. Palermo. *Compiler Techniques for Optimizing Communication and Data Distribution for Distributed-Memory Multicomputers*. PhD thesis, Dept. of Electrical and Computer Eng., Univ. of Illinois, Urbana, IL, June 1996. CRHC-96-09/UILU-ENG-96-2215.
13. D. J. Palermo and P. Banerjee. Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Proc. of the 8th Work. on Langs. and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 392–406, Columbus, OH, Aug. 1995. Springer-Verlag. 1996.
14. C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors. In *Proc. of the 18th Int'l Conf. on Parallel Processing*, pages II:39–48, St. Charles, IL, Aug. 1989.
15. S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, Feb. 1995.
16. R. Triolet, F. Irigion, and P. Feautrier. Direct Parallelization of Call Statements. *Proc. of the ACM SIGPLAN Symp. on Compiler Construction*, 21(7):176–185, July 1986.
17. C. W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice Univ., Houston, TX, Jan. 1993. COMP TR93-199.

This article was processed using the LaTeX macro package with LLNCS style