# Advanced Code Generation for High Performance Fortran

Vikram Adve and John Mellor-Crummey

Department of Computer Science and
Center For Research on Parallel Computation,
Rice University, Houston, Texas, USA

**Summary.** For data-parallel languages such as High Performance Fortran to achieve wide acceptance, parallelizing compilers must be able to provide consistently high performance for a broad spectrum of scientific applications. Although compilation of regular data-parallel applications for message-passing systems have been widely studied, current state-of-the-art compilers implement only a small number of key optimizations, and the implementations generally focus on optimizing programs using a "case-based" approach. For these reasons, current compilers are unable to provide consistently high levels of performance. In this paper, we describe techniques developed in the Rice dHPF compiler to address key code generation challenges that arise in achieving high performance for regular applications on message-passing systems. We focus on techniques required to implement advanced optimizations and to achieve consistently high performance with existing optimizations. Many of the core communication analysis and code generation algorithms in dHPF are expressed in terms of abstract equations manipulating integer sets. This approach enables general and yet simple implementations of sophisticated optimizations, making it more practical to include a comprehensive set of optimizations in data-parallel compilers. It also enables the compiler to support much more aggressive computation partitioning algorithms than in previous compilers. We therefore believe this approach can provide higher and more consistent levels of performance than are available today.

## 1. Introduction

Data-parallel languages such as High-Performance Fortran (HPF) [29, 31] aim to make parallel scientific computing accessible to a much wider audience by providing a simple, portable, abstract programming model applicable to a wide variety of parallel computing systems. For such languages to achieve wide acceptance, it will be essential to have parallelizing compilers that provide consistently high performance for a broad spectrum of scientific applications. To achieve the desired levels of performance and consistency, compilers must necessarily exploit a wide variety of optimization techniques and effectively apply them to programs with as few restrictions as possible.

Engineering HPF compilers that provide consistently high performance for a wide range of programs is a challenging task. The data layout directives in an HPF program provide an abstract, high-level specification of maximal data-parallelism and data-access locality. The compiler must use this information to choose how to partition the computation among processors,

determine what data movement and synchronization is necessary, and generate code to implement the partitioning, communication and synchronization. Accounting for interactions and feedback among these steps complicates program analysis and code generation. To achieve high efficiency, optimizations must analyze and transform the program globally within procedures, and often interprocedurally as well.

The most widely studied sub-problem of data-parallel compilation is that of compiling "regular" data-parallel applications on message-passing systems. Data-parallel programs are known as "regular" if the mapping of each array's data elements to processors can be described by an affine mapping function and the array sections accessed by each array reference can be computed symbolically at compile time. Even within this class of applications, state-of-the-art commercial and research compilers do not consistently achieve performance competitive with hand-written code [16, 24]. Although many important optimizations for such systems have been proposed by previous researchers, current compilers implement only a small fraction of these optimizations, generally focusing on the most fundamental ones such as static loop partitioning based on the "owner-computes" rule [39], moving messages out of loops, reducing the number of data copies, and exploiting collective communication. Furthermore, even for these optimizations, most research and commercial data-parallel compilers to date [7, 10, 15, 16, 17, 19, 24, 32, 33, 35, 42, 45, 46] (including the Rice Fortran 77D compiler [24]) perform communication analysis and code generation for specific combinations of the form of references, data layouts and computation partitionings. While such "case-based" approaches can provide excellent performance where they apply, they will provide poor performance for cases that have not been explicitly considered. More importantly, case-based compilers require a relatively high development cost for each new optimization because the analysis and code generation for each case is handled separately; this makes it difficult to achieve wide coverage with optimizations, which in turn makes it difficult to offer consistently high performance.

In this paper, we describe techniques to address key code generation challenges that arise in a sophisticated compiler for regular data-parallel applications on message-passing systems. We focus on techniques required to implement advanced optimizations and to achieve consistently high performance with existing optimizations. With minor exceptions, these techniques have been implemented in the Rice dHPF compiler, an experimental research compiler for High Performance Fortran. Although this paper focuses on compilation techniques for regular problems on message-passing systems, the dHPF compiler is being designed to integrate handling for regular and irregular applications, and to target other architectures including shared-memory and hybrid systems.

The principal code generation challenges we address are the following:

- *Flexible computation partitionings:* Higher performance can be achieved if compilers can go beyond the widely-used owner-computes rule to support a much more flexible class of computation partitionings. More general computation partitionings require two key compiler enhancements. First, they require robust communication analysis techniques that are not limited to specific partitioning assumptions. Second, they also require sophisticated code generation techniques to guarantee correctness in the presence of arbitrary control-flow, and to generate partitioned code with good scalar efficiency. The dHPF compiler supports a much more general computation partitioning (CP) model than in previous data-parallel compilers. We describe the communication analysis and code generation techniques required to support this model.
- *Robust algorithms for communication and code generation*: The core communication analysis, optimization, and code generation algorithms in dHPF are expressed in terms of abstract equations manipulating integer sets rather than as a collection of strategies for different cases. Optimizations we have formulated in this manner include message vectorization [14], message coalescing [43], recognizing in-place communication [1], code generation for our general CP model [1], non-local index set splitting [32], control-flow simplification [34], and generalized loop-peeling for improving parallelism. By formulating these algorithms in terms of operations on integer sets, we are able to abstract away the details of the CPs, references, and data layouts for each problem instance. All of these algorithms fully support our general computation partitioning model, and can be used for arbitrary combinations of computation partitionings, data layouts and affine reference subscripts.
- *Simplifying compiler-generated control-flow*: Loop splitting transformations, performed to minimize the dynamic cost of adding new guards, produce new loops with smaller iteration spaces which can render existing guards and loops inside redundant or infeasible. This raises the need for an algorithm that can determine the symbolic constraints that hold at each control point and use them to simplify or eliminate branches and loops in the generated code. We motivate and briefly describe a powerful algorithm for constraint propagation and control flow simplification used in the dHPF compiler [34]. In a preliminary evaluation, the algorithm has proven highly effective at eliminating excess control-flow in the generated code. Furthermore, we find that the general purpose control-flow simplification algorithm provides some or all of the benefits of special-purpose optimizations such as vector-message pipelining [43] and overlap areas [14].

Our aim in this chapter is to motivate and provide an overview of the techniques we use to address the challenges described above. The algorithms underlying these techniques are described and evaluated in detail elsewhere [1, 34]. In the following section, we use an example to describe

the basic steps in generating an explicit message-passing parallel program for HPF. We also use the example to show why integer sets are fundamental to the problem of HPF compilation, and describe the approaches used in previous data-parallel compilers for computing and generating code from integer sets. In Section 3 we describe a general integer-set framework underlying HPF compilation, and the implementation of this framework in dHPF. The framework directly supports integer set based algorithms for many of our optimizations, and these are briefly described in the subsequent sections. In Section 4, we define our general computation partitioning model and how we support code generation for it. In Section 5, we provide an overview of the principal communication optimizations in dHPF and then briefly describe the key optimizations that were formulated in terms of the integer set framework. In Section 6, we motivate and describe control-flow simplification in dHPF, and present a brief evaluation of its effectiveness. Finally, in Section 7, we conclude with a brief summary and discussion of the techniques described in this chapter.

## 2. Background: The Code Generation Problem For HPF

The High Performance Fortran standard describes a number of extensions to Fortran 90 to guide compiler parallelization for parallel systems. The language is discussed in some detail in an earlier chapter [29], and we assume the reader is familiar with the major aspects of the language (particularly, the data distribution directives). Throughout this paper, we assume a message-passing target system although many of the same analyses are required or profitable for shared-memory systems as well.

### 2.1 Communication analysis and code generation for HPF

To understand the basic problem of compiling an HPF program into an explicitly parallel message-passing program, and to motivate our use of a general integer-set framework for analysis and code generation, consider the simple example in Figure 2.1. The source loop represents a nearest-neighbor stencil computation similar to those found in partial differential equation solvers. The two arrays are aligned with each other and both are distributed `(block,block)` on a two-dimensional processor array. To generate an explicitly parallel code for the program, the compiler must first decide (a) how to partition the computation for each statement in the program, (b) which references might access non-local data due to the chosen partitioning, and (c) how and when to instantiate the communication to obtain this non-local data.

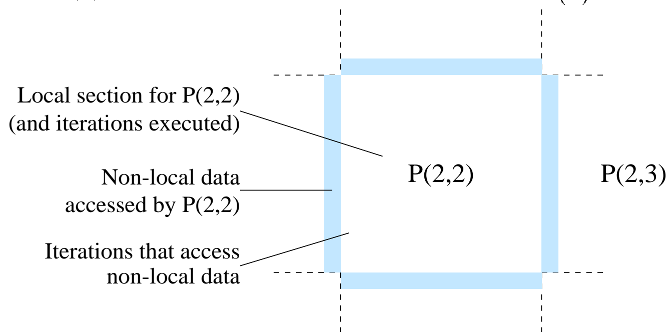
Assume the compiler chooses an “owner-computes” partitioning for the statement in the loop, i.e., each instance of the statement is executed by the

```
CHPF$ processors P(3,3)
CHPF$ distribute A(block,block) onto P
CHPF$ distribute B(block,block) onto P
      do 10 j=2,N-1
      do 10 i=2,N-1
        A(i,j) = 0.25*
                 (B(i-1,j)+B(i+1,j) +
                  B(i,j-1)+B(i,j+1))
  10  continue
```

(a) HPF source code

| P(1,1) | P(1,2) | P(1,3) |
|---|---|---|
| P(2,1) | P(2,2) | P(2,3) |
| P(3,1) | P(3,2) | P(3,3) |

(b) Processor array P(3,3)




(c) Communication and iteration sets

**Fig. 2.1.** Example illustrating integer sets required for code generation for Jacobi kernel

processors that own the value being computed, viz., `A(i,j)`. In this case, each of the four references on the right hand side (RHS) accesses some off-processor elements, namely the boundary elements on the four neighboring processors. Since the array `B` is not modified inside the loop nest, communication for these references can be moved out of the loops and placed before the loop nest.

In order to generate efficient explicitly parallel SPMD code, the compiler must compute the following quantities, and then use these to generate code. These quantities are illustrated in Figure 2.1:

1. the sections of each array allocated to (owned by) each processor;
2. the set of iterations to be executed by each processor (conforming with the owned section of array `A`);
3. the non-local data accessed from each other processor by each reference (the off-processor boundary sections shown in the figure);
4. the iterations that access non-local data and the iterations that access exclusively local data. (These sets are used by advanced optimizations such as those described in Section 5.3.)

All of these quantities can be symbolically represented as sets of integer tuples (representing array indices, loop iterations, or processor indices), or as mappings between integer tuples (e.g., an array layout is a mapping from processor indices to array indices). These sets and mappings are defined in Section 3. The sets may be non-convex, as is the set of iterations accessing non-local data shown in Figure 2.1.

To generate a statically partitioned, message-passing program, any data-parallel compiler must implicitly or explicitly compute the above sets, and then use these sets to generate code. The compiler typically generates SPMD code for a representative processor `myid` by performing the following tasks in some order (the resulting code is omitted here):

1. Synthesize a loop nest to execute the iterations assigned to `myid`.
2. For each message, if explicit buffering of data is necessary, synthesize code to pack a buffer at the sending processor and/or to unpack a buffer at the receiving processor.
3. Allocate storage to hold the non-local data, and modify the code to access data out of this storage (note that different references access non-local data in different iterations).
4. Allocate storage for the local sections for each array, and modify array references (for local data) to index appropriately into these local sections.

### 2.2 Previous Approaches to Communication Analysis and Code Generation

To compute the above sets and to generate code using them, the primary approach in most previous research and commercial compilers has been to focus on individual common cases and to precompute the iteration and communication sets symbolically for specific forms of references, data layouts and computation partitionings [7, 10, 15, 16, 17, 19, 24, 32, 33, 42, 46]. For example, to implement the Kali compiler [32], the authors pre-compute symbolically the iteration sets and communication sets for subscripts of the form $c$, $i+c$ and $c-i$, where $i$ is a loop index variable for BLOCK and CYCLIC distributions. The Fortran 77D compiler also handled the same classes of references and distributions, but computed special-case expressions for the iteration sets for "interior" and "boundary" processors [24]. (In fact, both these groups have described basic compilation steps in terms of abstract set operations [23, 32]; however, this was used only as a pedagogical abstraction and the corresponding compilers were implemented using case-based analysis.) Li and Chen describe algorithms to classify communication caused by more general reference patterns (assuming aligned arrays and the owner-computes rule), and generate code to realize these patterns efficiently on a target machine [33]. In general, these compilers focus on providing specific optimizations aimed at cases that are considered to be the most common and most important. The principal benefits of such case-based strategies are

that they are conceptually simple and hence lend themselves well to initial implementations, they have predictable and fast running times, and they can provide excellent performance in cases where they apply.

Three groups have used a more abstract and general approach based on linear inequalities to support code generation for communication and iteration sets [2, 3, 4, 5]. In this approach, each code generation or communication optimization problem is described by a collection of linear inequalities representing integer sets or mappings. Fourier-Motzkin elimination [41] is used to simplify the resulting inequalities, and to compute a range of values for individual index variables that together enumerate the integer points described by these inequalities. Code generation then amounts to generating loops to iterate over these index ranges. In the PIPS and Paradigm compilers, these techniques were primarily used for code generation for communication and iteration sets [3, 5]. In the SUIF compiler, these techniques were also applied to carry out specific optimizations including message vectorization, message coalescing (limited to certain combinations of references) and redundant message elimination [2].

The advantage of using linear inequalities over case-based approaches is that each optimization or code generation problem can be expressed and solved in abstract terms, independent of the specific forms of references, data layouts, and computation partitionings. Furthermore, Fourier-Motzkin elimination is applicable to arbitrary affine references and data layouts. The primary limitation of linear-inequality based approaches in previous compilers is that they have limited their focus to problems that can be represented by the intersection of a single set of inequalities. This limited the scope of their techniques so that, for example, they would be unable to support our general computation partitioning model, coalescing communication for arbitrary affine references, or general loop-splitting into local and non-local iterations. We considered all of these capabilities to be important goals for the dHPF compiler. A second drawback of linear-inequality based approaches is that each problem or optimization is expressed directly in terms of large collections of inequalities which must be constructed to represent operations such as intersections and compositions of sets and mappings. It appears much easier and more intuitive to express complex optimizations directly in terms of sequences of abstract set operations on integer sets, as shown in [1].

There is also a large body of work on techniques to enumerate communication sets and iteration sets in the presence of cyclic($k$) distributions (e.g., [12, 17, 28, 35, 45]). Compared to more general approaches based on integer sets or linear inequalities, these techniques likely provide more efficient support for cyclic($k$) distributions, particularly when $k > 1$, but would be much less efficient for simpler distributions, and are much less general in the forms of references and computation partitionings they could handle. Our goal has been to base the dHPF compiler on a general analysis framework that provides good performance in the vast majority of common cases

(within regular distributions), and requires such special-purpose techniques as infrequently as possible. Such techniques can be added as special-purpose optimizations in conjunction with the integer-set framework, but even in the absence of these techniques, we expect that the set framework itself will provide acceptably efficient support for cyclic($k$) distributions.

To summarize, we believe there are two essential advantages and one significant disadvantage of the more general and abstract approaches based on linear inequalities or integer-sets, compared with the case-based approaches. First, the former use general algorithms that handle the entire class of regular problems fairly well, whereas case-based approaches apply more narrowly and must fall back more often on inefficient techniques such as run-time resolution for cases they do not handle. In the absence of special-case algorithms, the general approaches are likely to provide much higher performance. Support for exploiting special-cases (e.g., for using collective communication primitives) can be added to the the former if they provide substantial performance improvements, but they should be needed in very few cases. Second, the more abstract framework provided by linear inequalities (to some extent) and by integer sets (to a greater extent) greatly simplifies the compiler-writer's task of implementing important optimizations that are generally applicable, and therefore make it *practical* to achieve high performance for a wider class of programs. By combining both generality and simplicity, we believe an approach such as that of using integer sets can provide higher and more consistent levels of performance than is available today. In contrast, the principal advantages of case-based approaches are that preliminary implementations can be simple, and that they typically have fast and predictable running times. For more general approaches, running time is the greatest concern since manipulation of linear inequalities and integer sets can be costly and unpredictable in difficult cases. This issue is discussed in more detail in later sections.

## 3. An Integer Set Framework for Data-Parallel Compilation

As discussed in the previous section, any compiler for a data-parallel language based on data distributions can be viewed as operating in terms of some fundamental sets of integer tuples and mappings between these sets. This formulation is made explicit in dHPF and in the SUIF compiler [2], and similar formulations have been discussed elsewhere [32, 23]. The integer set framework in dHPF includes the representation of these primitive quantities as integer tuple sets and mappings, together with the operations to manipulate them and generate code from them. The core optimizations in dHPF are implemented directly on this framework. This section explains the primitive components of the framework, and the implementation of the framework. The following sections describe the optimizations formulated using the framework.

### 3.1 Primitive Components of the Framework

An integer $k$-tuple is a point in $\mathcal{Z}^k$; a tuple space of rank $k$ is a subset of $\mathcal{Z}^k$. Any compiler for a data-parallel language based on data distributions operates primarily on three types of tuple spaces, and the three pairwise mappings between these tuple spaces [32, 23, 2]. These are:[1]

| | |
|---|---|
| $data_k$ : | index set of an array of rank $k, k \geq 0$ |
| $loop_k$ : | iteration space of a loop nest of depth $k, k \geq 0$ |
| $proc_k$ : | processor index space in a processor array of rank $k, k \geq 1$ |
| $Layout$ : | $\{[\underline{p}] \rightarrow [\underline{a}] : \underline{p} \in proc_n$ owns array element $\underline{a} \in data_k\}$ |
| $Ref$ : | $\{[\underline{i}] \rightarrow [\underline{a}] : \underline{i} \in loop_k$ references array element $\underline{a} \in data_k\}$ |
| $CPMap$ : | $\{[\underline{p}] \rightarrow [\underline{i}] : \underline{p} \in proc_n$ executes statement instance $\underline{i} \in loop_k\}$ |

Scalar quantities such as a "data set" for a scalar, or the "iteration set" for a statement not enclosed in any loop are handled uniformly within the framework as tuples of rank zero.[2] For example, the computation partitioning for a statement (outside any loop) assigned to processor $P$ in a 1-D processor array would be represented as the mapping $\{[\,] \rightarrow [p] : p = P\}$. Hereafter, the terms "array" and "iterations of a statement" should be thought of as including scalars and outermost statements as well. Note that any mapping we require, including a mapping with domain of rank 0, will be invertible.

Of these primitive sets and mappings, the sets *loop* and *proc* and the mappings *Layout* and *Ref* are constructed directly from the compiler's intermediate representation, and form the primary inputs for further analyses. These quantities are constructed from a powerful symbolic representation used in dHPF, namely global value numbering. A value number in dHPF is a handle for a symbolic expression tree. Value numbers are constructed from dataflow analysis of the program based on its Static Single Assignment (SSA) form [13], such that any two subexpressions that are known to have identical runtime values are assigned the same value number [21]. Their construction subsumes expression simplification, constant propagation, auxiliary induction variable recognition, and computing range information for expressions of loop index variables. A value number can be reconstituted back into an equivalent code fragment that represents the value.

Figure 3.1 illustrates simple examples of the primitive sets and mappings for an example HPF code fragment. For clarity, we use different set variables to denote points in different tuple spaces. The construction of the *Layout* mapping follows the two steps used to describe an array layout in HPF [31],

---

[1] We use names with lower-case initial letters for tuple sets and upper-case letters for mappings respectively.

[2] A set of rank 0, $\{[] : f(v_1, \ldots, v_n)\}$, should be interpreted as a boolean that takes the values true or false, depending on whether the constraints given by $f(v_1, \ldots, v_n)$ are satisfied. Here $v_1 \ldots v_n$ are symbolic integer variables.

```
real A(0:99,100), B(100,100)
processors P(4)
template T(100,100)
align A(i,j) with T(i+1,j)
align B(i,j) with T(*,i)
distribute t(*,block) onto P

read(*), N
do i = 1, N
  do j = 2, N+1
    A(i,j) = B(j-1,i)    ! ON_HOME  B(j-1,i)
  enddo
enddo
```

symbolic N

$$
\begin{aligned}
Align_A &= \{[a_1, a_1] \rightarrow [t_1, t_2] : t_1 = a_1 + 1 \wedge t_2 = a_2\} \\
Align_B &= \{[b_1, b_2] \rightarrow [t_1, t_2] : t_2 = b_1\} \\
Dist_T &= \{[t_1, t_2] \rightarrow [p] : 25p + 1 \leq t_2 \leq 25(p+1) \wedge 0 \leq p \leq 3\} \\
Layout_A &= Dist_T^{-1} \circ Align_A^{-1} \\
&= \{[p] \rightarrow [a_1, a_2] : max(25p+1, 1) \leq a_2 \leq min(25p+25, 100) \wedge \\
&\qquad 0 \leq a_1 \leq 99\} \\
Layout_B &= Dist_T^{-1} \circ Align_B^{-1} \\
&= \{[p] \rightarrow [b_1, b_2] : max(25p+1, 1) \leq b_1 \leq min(25p+25, 100) \wedge \\
&\qquad 1 \leq b_2 \leq 100\} \\
loop &= \{[l_1, l_2] : 1 \leq l_1 \leq N \wedge 2 \leq l_2 \leq N+1\} \\
CPRef &= \{[l_1, l_2] \rightarrow [b_1, b_2] : b_2 = l_1 \wedge b_1 = l_2 - 1\} \\
CPMap &= Layout_B \circ CPRef^{-1} \bigcap_{range} loop \\
&= \{[p] \rightarrow [l_1, l_2] : 1 \leq l_1 \leq min(N, 100) \wedge \\
&\qquad max(2, 25p+2) \leq l_2 \leq min(N+1, 101, 25p+26)\}
\end{aligned}
$$

**Fig. 3.1.** Construction of primitive sets and mappings for an example program. $Align_A$, $Align_B$, and $Dist_T$ also include constraints for the array and template ranges, but these have been omitted here for brevity.

namely, alignment of the array with a template and distribution of the template on a physical processor array (the template and processor array are each represented by a separate tuple space). The ON_HOME CP notation and construction of $CPMap$ are described in Section 4.

### 3.2 Implementation of the Framework

Expressing optimizations in terms of this framework requires an integer set package that supports all of the key set and map operations including intersection, union, difference, domain, range, composition of a map with another

map or set, and projection to eliminate a variable from a map or set. We use the Omega library developed by Pugh et al. at the University of Maryland for this purpose [27]. The library operations use powerful algorithms based on Fourier-Motzkin elimination for manipulating integer tuple sets represented by Presburger formulae [37]. In particular, the library provides two key capabilities: it supports a general class of integer set operations including set union, and it provides an algorithm to generate efficient code that enumerates points in a given sequence of iteration spaces associated with a sequence of statements in a loop [26]. (Appendix A describes this code generation capability.) These capabilities are an invaluable asset for implementing set-based versions of the core HPF compiler optimizations as well as enabling a variety of interesting new optimizations, described in later sections.

One potentially significant disadvantage of using such a general representation is the compile-time cost of the algorithms used in Omega. In particular, simplification of formulae in Presburger arithmetic can be extremely costly in the worst-case [36]. Pugh has shown, however, that when the underlying algorithms in Omega (for Fourier-Motzkin elimination) are applied to dependence analysis, the execution time is quite small even for complex constraints with coupled subscripts and also for synthetic problems known to cause poor performance [37]. These experimental results at least provide evidence that the basic techniques could be practical for use in a compiler. In dHPF, the Omega library has already proved to be a powerful tool for prototyping advanced optimizations based on the integer set framework. On small benchmarks, the compiler provides acceptably fast running times, for example, requiring about 3 minutes on a SparcStation-10 to compile the Spec92 benchmark Tomcatv with all optimizations. Further evidence for a variety of real applications will be required to judge whether or not this technology for implementing the integer set framework will prove practical for commercial data-parallel compilers. The dHPF implementation will provide a testbed for developing this evidence. If this approach does not prove practical, it is still possible that a simpler and more efficient underlying set representation could be used to support the same abstract formulation of optimizations, but with some loss of precision.

Another significant and fundamental limitation is that Presburger arithmetic is undecidable in the presence of multiplication. For this reason, the Omega library provides only limited support for handling multiplication, and in particular, cannot represent sets with an unknown (i.e., symbolic) stride. Most importantly (from the viewpoint of HPF compilation), such strided sets are required for any HPF distribution when the number of processors is not known at compile time, and for a cyclic($k$) distribution with unknown $k$. We have extended our framework to permit these parameters to be symbolic, as described below. Symbolic strides also arise for a loop with a non-constant stride or a subscript expression with a non-constant coefficient, although we expect these to be rare in practice. These are not supported by our frame-

work, and would have to fall back on more expensive run-time techniques such as a finite-state-machine approach for computing communication and iteration sets (for example, [28]), or an inspector-executor approach.

To permit a symbolic number of processors or cyclic($k$) distribution with symbolic $k$, we use a virtual processor (VP) model that naturally matches the semantics of templates in HPF [22]. The VP model uses a virtual processor array for each physical processor array, using template indices (i.e., ignoring the distribute directive) in dimensions where the block size or number of processors is unknown, but using physical processor indices in all other dimensions. Using physical processor indices where possible facilitates better analysis and improves the efficiency of generated code. All of the analyses described in the following sections operate unchanged on physical or virtual processor domains. During code generation for each specific problem (e.g., generating a partitioned loop), we add extra enclosing loops that iterate over the VPs that are owned by the relevant physical processor (e.g., the representative processor `myid`). For each problem, we use an additional optimization step (consisting of a few extra integer set equations) to compute the precise set of iterations required for these extra loops, and therefore to minimize the runtime overhead in the resulting code. The details of our extensions to handle a symbolic number of processors are given in [1].

## 4. Computation Partitioning

A computation partitioning (CP) for a statement is a precise specification of which processor or processors must execute each dynamic instance of the statement. The CPs chosen by the compiler play a fundamental role in determining the performance of the resulting code. For the compiler to have the freedom to choose a partitioning well-suited to an application's needs, the communication analysis and code generation phases of the compiler must be able to support a flexible class of computation partitionings. In this section, we describe the computation partitioning model provided by dHPF and the code generation framework used to support the model. The communication analysis techniques supporting the model are described in Section 5.

### 4.1 Computation Partitioning Models

Most research and commercial compilers for HPF to date primarily use the *owner-computes* rule [39] to partition a computation. This rule specifies that each value assigned by a statement is computed by the owner (i.e., the "home") of the location being assigned the value, e.g., the left hand side (LHS) in an assignment. The owner-computes rule amounts to a simple heuristic choice for partitioning a computation. It is straightforward to show that this approach is not optimal in general [12]. An alternate partitioning

strategy used by SUIF [2] and Barua, Kranz & Agarwal [6] requires a CP to be described by a single affine mapping of iterations to processors, and assigns a single CP to an entire loop iteration and not to individual statements in a loop. This strategy is also not optimal, because (in general) it does not permit optimal CPs to be chosen separately for different statements in a loop.

A major goal of the dHPF compiler is to support more general computation partitionings. Doing so requires new support from the compiler's communication analysis and code-generation phases. Previous compilers based on the owner-computes rule have benefited from two key simplifying assumptions: (1) communication patterns are defined by a single pair of LHS and right-hand-side (RHS) references, and (2) all communication is caused by *reads* of non-local data. The SUIF partitioning model also has the benefit that each communication is defined by a single reference and a single CP mapping (before coalescing communication), although write references can cause communication too. This model has the additional benefit that code generation is greatly simplified by having a common CP for all statements in a loop (which will become clear from the discussion in Section 4.2). None of these simplifying assumptions are true for the more general partitioning model used in dHPF.

**4.1.1 The Computation Partitioning Model in dHPF.** The computation partitioning model supported by dHPF combines and generalizes the features of both previous CP models described above (the owner-computes rule and the SUIF model). Below we describe the key features of the dHPF CP model including the implicit CP representation used by early phases in the compilation and conversion to an explicit CP representation required for communication analysis and code generation. In Section 4.2.1, we discuss code generation for these general CPs and in Section 5 we discuss the role of CPs in communication analysis.

In dHPF, a computation partitioning for a statement can be specified as the set of owners of the locations accessed by one *or more* arbitrary data references, Every statement (including control flow statements) can be assigned a partitioning independent of every other statement, restricted only to preserving the semantics of the source program. For a statement $S$ enclosed in a loop nest with iteration space $\underline{i}$, the CP of $S$ is specified by a union of one or more ON_HOME terms:

$$CP(S) : \bigcup_{k=1}^{k=n} \text{ON_HOME } A_k(f_k(\underline{i})) \quad (4.1)$$

An individual term ON_HOME $A_k(f_k(\underline{i}))$, specifies that the instance of $S$ in iteration $\underline{i}$ is to be executed by the processor(s) that own the array element(s) $A_k(f_k(\underline{i}))$. This set of processors is uniquely specified by subscript vector $f_k(\underline{i})$

and the layout of array $A_k$ at that point in the execution of the program.[3] This *implicit* representation of a computation partitioning supports arbitrary index expressions or any set of values in each index position in $f_k(\underline{i})$. A data reference $A_k(f_k(\underline{i}))$ in an ON_HOME clause need not be a reference existing in the program. Even the variable $A_k$ and its corresponding data layout may be synthesized for representing a desired CP, though our implementation is restricted to legal HPF layouts. With this representation, the CP model permits specification of a very wide class of partitionings.

While early analysis phases in the dHPF compiler use this implicit CP representation, communication analysis and code generation require that the CP for each statement is converted into an *explicit* mapping of type $CPMap$ defined in Section 3.1. The integer set framework is used to construct this explicit mapping. This construction requires that each subscript expression in $f_k(\underline{i})$ be an affine expression of the index variables, $\underline{i}$, with known constant coefficients, or a strided range specifiable by a triplet *lb*:*ub*:*step* with known constant *step*. We construct the explicit integer tuple mapping representing the CP for a statement as follows.

$$CPMap(S) = \bigcup_{k=1}^{k=n} (Layout_{Ak} \circ Ref_k^{-1}) \bigcap\nolimits_{range} loop. \tag{4.2}$$

For each term ON_HOME $A_k(f_k(\underline{i}))$ in $CP(S)$, the composition of its layout and inverse reference maps results in a new map that specifies all possible iterations assigned to each processor by this CP term. We restrict the range of this map to the iteration space given by *loop*. Taking the union over all CP terms gives the full mapping of iterations to processors for statement $S$. $CPMap(S)$ specifies the processor assignment for the single instance of statement $S$ in loop iteration $\underline{i}$. Figure 3.1 shows a simple example of the construction of $CPMap$. The mapping can be vectorized over the range of iterations of one or more enclosing loops to represent the combined processor assignment for the set of statement instances in those loop iterations.

Careful assignment of CPs to control-flow related statements (namely, `DO`, `IF`, and `GOTO` statements, as well as labeled branch targets) is necessary to preserve the semantics of the source program. In particular, a *legal partitioning* must ensure that each statement in the program is reached by a superset of the processors that need to participate in its execution, as specified by its CP. The code generation phase will then ensure that the statement is executed by exactly the processors specified by the CP. The algorithms dHPF uses to select computation partitions and ensure legality are beyond the scope of this paper. In Section 4.2.1, we discuss the interaction between correctness

[3] In the presence of dynamic REALIGN and REDISTRIBUTE directives, we assume that only a single known layout is possible for each reference in the program. Multiple reaching layouts would require generating multi-version code or assuming that the layout is unknown until run time (as done for inherited layouts).

constraints on CP assignments for control-flow related statements and code generation.

To support dHPF's general computation partitioning model, the communication analysis and code-generation phases in the compiler must fully support any legal partitioning. Supporting this partitioning model would be impractical using a case-based approach; the dHPF compiler's representation of computation partitionings using an abstract integer set framework has proven essential for making the required analysis and code generation capabilities practical.

### 4.2 Code Generation to Realize Computation Partitions

A general CP model such as that in dHPF poses several challenges for static code generation. First, the code generator must ensure correctness in the presence of arbitrary structured and unstructured control flow, without sacrificing available parallelism. Second, generating efficient parallel code for a loop nest containing multiple statements that potentially have different iteration spaces is an intrinsically difficult problem. Previous compilers use simple approaches for code generation and do not solve this problem in its general form (as described briefly below), but Kelly, Pugh & Rosser have developed an aggressive algorithm for "multiple-mappings code generation" which directly tackles this problem [26]. A third and related difficulty, however, is that good algorithms for generating efficient code (like that of Kelly, Pugh & Rosser) will be inherently expensive because of the potential complexity of the iteration spaces and the resulting code. Ensuring reasonable compile times requires an effective strategy to control the compile-time cost of applying such an algorithm while still producing as high quality code as possible. A fourth problem (not directly related to a general CP model) is that static code generation techniques will not be useful for a code with irregular or complex partitionings. Such cases require runtime strategies such as the inspector-executor approach (e.g., [32, 18, 40]). However, regular and irregular partitionings may coexist in the same program, and perhaps even in a single loop nest. This raises the need for a flexible code generation framework that allows each part of the source program to be partitioned using the most efficient strategy applicable.

Before describing the techniques used in dHPF to address these challenges, we briefly describe the strategies used to realize computation partitions in current compilers, and the limitations of these strategies in addressing these challenges. We begin with the second of the four problems described above because the approaches to addressing this problem have implications for the other problems as well. As in the rest of the paper, we focus on compile-time techniques that are applicable when the compiler can compute static (symbolic) mappings between processors and the data they own or communicate. If a static mapping is not computable at compile time, alternative

strategies that can be used include run-time resolution [39], the inspector-executor approach, and run-time techniques for handling `cyclic(k)` partitionings. The latter two approaches are described in other chapters within this volume [40, 38].

It is relatively straightforward to partition a simple loop nest that contains a single statement or a sequence of statements with the same CP. The loop bounds can be reduced so that each processor will execute a smaller iteration space that contains only the statement instances the processor needs to execute. For loops containing multiple statements with different CPs, it is important that each processor execute as few guards as possible to determine which statement instances it must execute in each iteration. Previous compilers, such as IBM's pHPF compiler[16], rely on loop distribution to construct separate loop nests containing statements with identical CPs, so as to avoid the need for run-time guards. There are two drawbacks to using loop distribution in this manner. First, loop distribution may be impossible because of cyclic data dependences in the loop. In such cases, compilers add statement guards to implement the CPs and, except for Paradigm, don't reduce loop bounds. The Paradigm compiler reduces loop bounds to the convex hull of the iteration spaces of statements inside the loop, in order to reduce the number of guards executed [5]. Second, fragmenting a loop nest into a sequence of separate loops over individual statements can (a) significantly reduce reuse of cached values between statements, and (b) significantly increase the contribution of loop overhead to overall execution time. A loop fusion pass that attempts to recover cache locality and reduce loop bound overhead is possible, but complex. Both the IBM and the Portland Group HPF compilers use a loop fusion pass, but apply it only to the simplest cases, namely conformant loops [10, 16].

Kelly, Pugh & Rosser describe an aggressive algorithm to generate efficient code without relying on loop distribution, for a loop nest containing multiple statements with different iteration spaces [26]. Given a sequence of (possibly non-convex) iteration spaces, the algorithm, MMCODEGEN, synthesizes a code fragment to enumerate the points in the iteration spaces in lexicographic order, tiling the loops as necessary to lift guards out of one or more levels of loops. Thus, the algorithm provides one of the key capabilities required to support our general CP model. The algorithm is briefly described in Appendix A along with an example that highlights its capabilities.

As mentioned earlier, one potential drawback of an algorithm like MMCODEGEN is that it can be costly for large loop nests with multiple iteration spaces. Because of the potential for achieving high performance, however, we use the algorithm in dHPF as one of the core techniques for supporting the general CP model, and develop a strategy to control compile-time cost while still producing high quality code. To our knowledge, this is the first use of their algorithm for code generation in a data-parallel compiler, and therefore these issues have not been addressed previously.

The techniques used in previous compilers to partition code in the presence of control-flow are not clearly described in the written literature. Some compilers simply replicate the execution of control-flow statements on all processors, which is a simple method to ensure correctness but can substantially reduce parallelism in loops containing control flow [19]. Many other compilers ignore control flow because loop bounds reduction and ownership guards will enforce the appropriate CP for enclosed statements. However, this approach sacrifices potential parallelism for the sake of simplifying code generation. In particular, by not enforcing an explicit CP for a block structured `IF` statement, all processors that enter the scope enclosing the `IF` will execute the test, enter the appropriate branch, and execute CP guards for the statements inside, even though some of the processors do not need to participate in the execution of the enclosed statements at all.

**4.2.1 Realizing Computation Partitions in dHPF.** We have developed a hierarchical code generation framework to realize the general class of partitionings that dHPF supports. Our approach is hierarchical because it operates on nested block structured scopes, one scope at a time. (A *scope* refers to a sequence of statements immediately enclosed within a procedure, a `DO` loop, or a single branch of an `IF` statement.) Key benefits of the hierarchical code generation framework are (1) it supports partitioning of scopes that can contain arbitrary control flow, (2) it supports multiple code generation strategies for individual scopes so that each scope can be partitioned with the most appropriate strategy, and (3) it uses a two pass strategy that is effective at minimizing guard overhead without sacrificing compile-time efficiency. Below, we first describe the hierarchical code generation framework, and then describe strategies used in dHPF to generate code for a single scope within this framework.

*The hierarchical code generation framework.* Briefly, the code generation framework in dHPF operates as follows. Each scope in the program is handled independently. The code generation operates one scope at a time, visiting scopes bottom-up (i.e., innermost scopes first). Although the framework operates one scope at at time, any particular strategy can partition multiple scopes in a single step if desired. At each scope that has not yet been partitioned, the framework attempts to apply a sequence of successively more general strategies until one successfully performs the partitioning for the scope.

Control flow in the program is handled as follows. This discussion assumes that a pre-processing phase has transformed all `DO` loops into `DO/ENDDO` form and and relocated all branch target labels to `CONTINUE` statements. The compiler computes partitionings for control-flow statements so that all processors that need to execute any statement reach it, but as few processors execute the control-flow statements as possible in order to maximize parallelism. Informally, the computation partitioning for an `IF` statement or a `DO` loop must involve all of the processors that need to execute any statement that is transitively control-dependent on it. For an `IF` statement, the union of the CPs

of its control dependents gives a suitable CP. For a `DO` loop, the the union of the CPs of its control dependents gives a CP suitable for a single iteration; a suitable CP for the entire compound `DO` statement can be computed by vectorizing the iteration CP over the loop range. A GOTO statement is assigned the union of the CPs of all control-flow statements on which it is immediately control-dependent. Note that this will be a superset of the CPs of the statements following the branch target. i.e., any processors that need to execute the target statements will reach those statements (the condition controlling the branch is satisfied). Finally, a branch target label is assigned the CP of the immediately enclosing scope, for reasons discussed below.

In order to ensure correctness while preserving maximum parallelism in the presence of control-flow, the code generation framework simply has to ensure that the above assignment of CPs to control-flow related statements is correctly preserved. In particular, any particular strategy used to partition a scope must correctly enforce the CPs of all statements *within* the scope. The only subtlety is that reducing the bounds of a DO loop does not enforce the CP of the compound DO loop statement itself; that must be enforced when partitioning the scope enclosing the DO loop (otherwise, extra processors may evaluate the bounds of the DO loop).

The above handling of partitioned control flow statements yields a key simplification of the framework, namely, that each block-structured scope can be handled independently, even in the presence of arbitrary control flow such as a GOTO that transfers control from one scope to another. In particular, the CPs assigned to individual GOTO statements is simply enforced during code generation, independent of the location and CPs of its branch targets. A significant difficulty that must be addressed is that GOTOs are matched with the correct branch targets (and labels). This can be difficult because statements in different scopes and even statements in the same scope may be cloned into different numbers of copies. (Statements can be cloned by the tiling performed by MMCODEGEN to minimize guards, as shown in the example in Appendix A.) We ensure that branches and branch targets are matched, as follows.

Fortran semantics dictate that a GOTO cannot branch from an outer to an inner scope; a GOTO can only branch within the same scope or to an outer scope. The CP assigned to a GOTO may cause it to be cloned into multiple copies. By assigning a labeled statement the CP of its enclosing scope, we ensure that a label appears in every instance of that scope (in particular, in a DO scope, every iteration of the enclosing scope will include the labeled CONTINUE). Since every GOTO branching to this label must come from the same or an inner scope, every cloned copy of the GOTO will be matched with exactly one copy of the labeled CONTINUE. Furthermore, the GOTOs that must match a particular copy of the label are exactly those that appear within the same instance of the scope enclosing the label. This allows us to to

renumber the label definitions and any matching label references. The details of the renumbering scheme are described elsewhere [1].

The second key issue we address with the framework is controlling the compile-time cost of using expensive algorithms such as MMCODEGEN, while still ensuring that guard overhead is minimized. There are two features in the framework that ensure efficiency. First, we take advantage of the independent handling of scopes to apply MMCODEGEN independently one loop or one perfect loop nest at a time. This ensures that the iteration spaces in each invocation of MMCODEGEN are as simple as possible (though there can still be multiple different iteration spaces). Second, the use of a bottom-up approach greatly reduces the number of times MMCODEGEN is invoked, compared to a top-down approach. The drawback however, is that the top-down approach could yield much more efficient code. This tradeoff between the bottom-up and top-down approaches arises as follows.

When generating code for a `DO` scope, the loop's iteration space is often split into multiple sections to enable guards to be lifted out of the loop. If we used a top-down scope traversal order for generating code, information about the bounds of the different sections could be passed downward during code generation and exploited in inner scopes. However, a top-down strategy would require many more applications of the partitioning algorithm than a bottom-up strategy. For example, for a triply nested loop in which each loop will be split into two sections by code generation, a top-down strategy would invoke the loop partitioning algorithm seven times. A bottom-up strategy would invoke it only three times. Because of the potentially high compile-time cost of the former, we use a bottom-up code generation strategy.

We use two techniques to ensure the quality of the generated code despite the trade-offs made above. First, an important optimization we apply when generating code for individual scopes is to exploit as much known contextual information as possible about the enclosing scopes. Second, we use a powerful, global control-flow simplification phase as the last step of code generation, which further simplifies the control-flow in the resulting program. The control-flow simplification algorithm is described in section 6. The use of available contextual information during the bottom-up strategy is described below. Together, these achieve much or all of the benefit of a top-down code generation strategy in which full context information is available to code generation in inner scopes, but at a fraction of the cost.

When generating code for a scope in the bottom-up strategy, we can assume that code generation in the enclosing scope will ensure that only the correct processors enter the current scope. For example, consider the loop nest in Fig. 4.1. Statements `s1`, `s2`, and `s3` each have a simple computation partition consisting of a single ON_HOME clause. $LoopCP_j(i)$, represents the CP for the $j$ loop, which consists of the union of the CPs of the statements in its scope vectorized across the range of the $j$ loop. Inside the $j$ loop, we assume that the constraints in $LoopCP_j(i)$ hold because these constraints

```
do i = 1, N
  S1(i)
  do j = 1, M
    S2(i,j)
    S3(i,j)
  enddo
enddo
```

$$
\begin{aligned}
LoopCP_i() &= CP_1(1:N) \cup LoopCP_j(1:N) \\
CP_1(i) &= \{ [i] : \text{proc. myid owns } A_1(f_1(i)) \} \\
LoopCP_j(i) &= CP_2(i, 1:M) \cup CP_3(i, 1:M) \\
CP_2(i,j) &= \{ [i,j] : \text{proc. myid owns } A_2(f_2(i,j)) \} \\
CP_3(i,j) &= \{ [i,j] : \text{proc. myid owns } A_3(f_3(i,j)) \}
\end{aligned}
$$

**Fig. 4.1.** Example showing iteration sets constructed for code generation.

will have been enforced when partitioning the enclosing $i$ loop. Any code generation strategy used for the inner scope can exploit this information. Similarly, $LoopCP_i()$ represents the CP for the $i$ loop, which consists of the union of the CPs of the statements it contains, vectorized across the range of the $i$ loop. We assume that the constraints in $LoopCP_i()$ are true when generating code for the $i$ loop.

*Realizing a CP for a single scope.* The first step in the process for partitioning a scope is to separate the statements in the scope into *statement groups*, which are sequences of adjacent statements that have homogeneous computation partitions. Second, we use equation 4.2 to construct the *explicit* representation of the iteration space for each statement group according to its computation partitioning. Third, we use some available strategy to partition the computation in the scope. When multiple strategies are available, we currently apply them in a fixed sequence, stopping when one succeeds in partitioning the scope. This permits a fixed series of strategies to be tried (typically attempting specific optimizations and, if these fail, then finally applying some general strategy).

We currently support two strategies for partitioning a loop: bounds reduction, or loop-splitting combined with bounds reduction for the individual loop sections generated by splitting (the latter is described in Section 5.3). For non-loop scopes (conditional branches and the outermost routine level), we also use bounds reduction which reduces to inserting guards on the relevant statements. Two alternatives applicable to loops or statement groups with irregular data layouts or irregular references are under development, namely, runtime resolution and an inspector-executor strategy.

To perform bounds reduction as part of the above strategies, we apply Kelly, Pugh & Rosser's MMCODEGEN algorithm to the sequence of iteration spaces for the statement groups in a scope. Applying MMCODEGEN to the iteration spaces for statement groups reduces loop bounds as needed and lifts guards out of inner loops when statement groups with non-overlapping iteration spaces exist. This results in a code template template with placeholders representing the statement groups. Finally, we replace each of the placeholders in the code template by a copy of the code for the corresponding statement group. When labels are present in the code for the statement groups, we renumber the labels to ensure unique numbers as discussed earlier.

As an alternative to this base strategy for realizing the computation partition for a scope, Section 5.3 describes a loop splitting transformation that may be applied during code generation to any perfect loop nest that has no carried dependences. From the perspective of the computation partitioning code generation, this approach serves as an alternate partitioning method which subdivides the iteration space for a `DO` into a sequence of iteration spaces, and then generates code for each with the method described above for a single scope. The purpose of the splitting transformation is described in section 5.3.

Another much more specialized strategy we expect to add for loop nests is a code transformation for coarse-grain pipelining. The transformation simultaneously performs strip-mining of one or more non-partitioned loops and loop bounds reduction for partitioned loops. The last two optimizations (loop-splitting and pipelining) illustrate that the hierarchical code generation framework provides a natural setting within which to perform any code transformation that has the side effect of producing partitioned code, i.e., realizing the CPs assigned to statements in a scope.

## 5. Communication Code Generation

On message-passing systems, the most efficient communication is obtained when the compiler can statically compute the set of data that needs to be exchanged between processors to satisfy each non-local reference. For references with statically analyzable communication requirements, a data-parallel compiler must compute the set of non-local data to be communicated for each non-local reference, and then use these sets to generate efficient code to pack, communicate, unpack, and access the non-local data. In this section, we describe implementation techniques for several key communication optimizations used in the dHPF compiler to synthesize high performance communication code for regular applications. Many of these techniques are based on the integer set framework. For references with unanalyzable communication requirements, typically due to non-affine subscripts, runtime techniques such as the inspector-executor model must be used to manage communication. For more information on such techniques, we refer the reader to Chapter 21 and the references therein.

The dHPF compiler includes a comprehensive set of communication optimizations that have been identified as important for high performance on message-passing systems. The benefits obtained from these optimizations (with very few exceptions) can vary widely between applications as well as between different systems. This implies that a compiler may have to incorporate many different optimizations to obtain consistently high performance across large classes of applications and systems. Previous commercial and research compilers, however, have generally implemented only a few of these techniques because of the significant implementation effort entailed in each

case. The important communication optimizations in the dHPF compiler include the following.

**Optimizations to reduce message overhead**

- *Message vectorization* moves communication out of loops in order to replace element-wise communication with fewer but larger messages. This is implemented by virtually all data-parallel compilers, but in case-based compilers it is usually restricted to specific reference patterns for which the compiler can derive (or conservatively approximate) the data sets to be communicated [16, 24, 32].
- *Exploiting collective communication* is essential for achieving good speedup in important cases such as reductions, broadcasts, and array redistribution [33]. On certain systems, collective communication primitives may also provide significant benefits for other patterns such as shift communication. The important patterns (particularly reductions and broadcast) have been supported in most data-parallel compilers.
- *Message coalescing* combines messages for multiple non-local references to the same or different variables, in order to reduce the total number of messages and to eliminate redundant communication. Previous implementations in Fortran 77D [24], SUIF [2], Paradigm [5], and IBM's pHPF [11, 16] have some significant limitations. In particular, coalescing can produce fairly complex data sets from the union of data sets for individual references. The previous implementations are limited to cases where the combined data sets are representable with (or can be approximated by) regular sections (in Fortran 77D, Paradigm and pHPF) or a single collection of inequalities (in SUIF).
- *Coarse-grain pipelining* trades off parallelism to reduce communication overhead in loop nests with loop-carried, cross-processor data dependences. It is an important optimization for effectively implementing parallelism in such loop nests because the only alternative may be to perform a full array redistribution, which can be much more expensive. To our knowledge, this optimization has been implemented in a few research compilers [24, 5] and one commercial one [16].

**Optimizations to overlap communication with computation**

- *Dataflow-based communication placement* attempts to hide the latency of communication by placing message sends early and receives late so as to overlap messages with unrelated computation. A few compilers including IBM's pHPF, SUIF, Paradigm, and Fortran 77D have used dataflow techniques to overlap communication in this manner.
- *Communication overlap via non-local index set splitting* attempts to overlap communication from a given loop nest with the local iterations of the same loop nest. This overlap generally cannot be achieved by the above dataflow placement techniques. Non-local index set splitting (or loop splitting) separates iterations that access non-local data from those that access

only local data. Communication can be overlapped with the local iterations by first executing send operations for the non-local data required in the loop, then the local iterations, then the receives, and finally the non-local iterations. Loop splitting was implemented in Kali [32], albeit with significant limitations as described in Section 5.3.

**Optimizations to minimize data buffering and access costs**

- *Minimizing buffer copying overhead* is essential to minimize the overall cost of communication. This can be achieved in multiple ways. First, in most message-passing implementations, when the data to be sent or received is contiguous in memory, it can be communicated "in-place" rather than copied to or from message buffers. Second, asynchronous send and receive primitives can be used to avoid additional buffer copies between user and system buffers by making user-level buffers available for the duration of communication. Third, non-local data received into a buffer can in some cases be directly referenced out of the buffer (if the indexing functions can be generated by the compiler), thus avoiding an unpacking operation. All of these techniques appear to be widely used in data-parallel compilers, though the effectiveness of the implementations may vary.
- *Minimizing buffer access checks via non-local index-set splitting.* Access checks (i.e., ownership tests) are required when the same reference may access local data from an array or non-local data from a separate buffer on different loop iterations. Loop-splitting separates out the local iterations (which are guaranteed to access local data) from the non-local ones. Even the latter may not need access checks if all non-local references now access only non-local data. The alternative to this transformation is to copy local and non-local data into a common buffer (as done in the IBM pHPF compiler [16]), which can be costly in time and memory usage. As mentioned above, non-local index set splitting was implemented in a limited form in Kali.
- *Overlap areas for shift communication* are extra boundary elements added to the local sections of arrays involved in shift communication [14]. They permit local and non-local data to be referenced uniformly, thus avoiding the need for the access checks (or alternatives) mentioned above. Generally, interprocedural analysis has to be used to determine the size of required overlap areas globally for each array. Simpler implementations may waste significant memory and may have to be controlled by the programmer. Overlap areas have been implemented in several research and commercial compilers.

The Rice dHPF compiler implements all of the above optimizations except coarse-grain pipelining and the use of asynchronous message primitives (both these are currently being implemented). Some other specific communication optimizations have been implemented in other compilers but are not included in dHPF. IBM's pHPF coalesces diagonal shift communication into

two messages [16], whereas dHPF requires three. This is useful, for example, in stencil computations that access diagonal neighbors such as a nine-point stencil over a two-dimensional array. Chakrabarti et al. describe a powerful communication placement algorithm that can be used to maximize opportunities for message coalescing or to balance message coalescing with communication overlap [11]. SUIF uses array dataflow analysis to communicate data directly from a processor executing a non-local write to the next processor executing a non-local read [2], whereas dHPF must use an extra message to send the data first to the owner and from there to the reader. The former two optimizations can be directly added to the current implementation of dHPF. The SUIF model is a different and significantly more complex communication model compared to that used in dHPF, and there is little evidence available so far to evaluate whether the additional complexity is justified for message-passing systems.

One reason that it has been practical to implement a fairly large collection of advanced optimizations in dHPF is our use of the integer set framework. By formulating optimizations abstractly in terms of integer set operations, we have obtained simple, concise, and general implementations of some of the most important phases of the compiler (such as communication code generation) as well as of complex optimizations like loop-splitting. These implementations broadly apply to arbitrary combinations of affine references, data distributions, and computation partitionings, because the analysis is not dependent on specific forms of these parameters. In the remainder of this section, we briefly describe the implementation of communication optimizations that use the integer set framework. These include our entire communication generation phase which incorporates message vectorization and coalescing, the two optimizations based on non-local index set splitting and an algorithm for recognizing in-place communication. A control-flow simplification algorithm, which is also implemented using integer sets, is described in Section 6.

### 5.1 Communication generation with message vectorization and coalescing

The communication insertion steps in dHPF can be classified into two phases: a preliminary decision-making phase that identifies and places the required communication in the program, and a communication generation phase that computes the sets of processors and data involved in each communication, and synthesizes code to carry out the communication. In this paper, we primarily focus on the communication generation phase which is based on the integer set framework. We briefly describe the decisions made in the former phase, since these directly feed in as inputs to communication generation.

The preliminary communication analysis steps in dHPF determine (a) which references are potentially "non-local", i.e., might access non-local data, (b) where to place communication for each reference, (c) whether to use a

collective communication primitive in each case, and (d) when communication for multiple references can be combined. The first step is a very simple analysis to filter out references that can easily be proven to access only local data. The second step uses a combination of dependence and dataflow analysis to choose the placement of communication so as to determine how far each message can be vectorized out of enclosing loops, and to optionally move communication calls early or late to hide communication latency [30]. The third step uses the algorithms of Li and Chen [33] to to determine if specialized collective communication primitives such as a broadcast could be exploited. (Reductions are recognized using separate algorithms.) Otherwise, the compiler directly implements the communication using pairwise point-to-point communication. The fourth step chooses references whose communication can be combined. Any two references whose communication involves one or more common pairs of processors can be coalesced in our implementation. In practice, however, it it is usually not beneficial to combine references that should use different communication primitives, such as a broadcast with any pairwise point-to-point communication. (One instance where it is profitable is combining a reduction and a broadcast by using a special reduction primitive like `MPI_AllReduce`, which leaves every processor involved with a copy of the result.) We refer to the entire collection of messages required for a set of coalesced references as a single *logical communication event*.

The code generation phase must then use the results of the previous phases to synthesize vectorized and coalesced messages that implement the desired communication for each logical communication event. For each reference, the compiler first computes the set of data to send between pairs (or groups) of processors; these communication sets depend on the reference, layout, computation partitioning, and the loop level at which vectorized communication is to be performed. Message coalescing requires computing the union of the above communication sets for the coalesced references. We directly compute the communication sets for each communication event using a sequence of integer set operations, independent of the specific form of the reference, layout, and computation partitioning. We then generate code from these sets directly.

The integer set equations used to compute the communication sets for each logical communication event are described in detail elsewhere [1]. We briefly describe the key aspects of the algorithm here. The goal of the algorithm is to compute two separate maps for a fixed symbolic processor index $\underline{m}$ (where $\underline{m}$ is the index tuple for processor `myid` in the processor array to which the data is mapped, and `myid` is the representative processor index of the SPMD program).

$$\begin{aligned} SendCommMap(\underline{m}) &= \{[\underline{p}] \rightarrow [\underline{a}] : \text{array elements } \underline{a} \text{ that proc. } \underline{m} \text{ must send to proc. } \underline{p}\ \} \\ RecvCommMap(\underline{m}) &= \{[\underline{p}] \rightarrow [\underline{a}] : \text{array elements } \underline{a} \text{ that proc. } \underline{m} \text{ must receive from proc. } \underline{p}\ \} \end{aligned}$$

```
DataAccessedMap =
 { [p1,p2]->[b1,b2] :
   max(1, 20*p1) <= b1 <= min(20*p1+19, 58) &&
   max(2, 20*p2+1) <= b2 <= min(20*p2+20, 59) };

nlDataAccessed({m1,m2}) =
 { [b1,b2] :
   1 <= m1 <= 2 && b1 = 20*m1 &&
   max(2, 20*m2+1) <= b2 <= min(20*m2+20, 59) };

SendCommMap({m1,m2}) =
 { [p1,p2]->[b1,b2] :
   p1 = m1+1 && p2 = m2 && 0 <= m1 <= 1 && b1 = 20*m1+20 &&
   max(2, 20*m2+1) <= b2 <= min(20*m2+20, 59) };

RecvCommMap({m1,m2}) =
 { [p1,p2]->[b1,b2] :
   p1 = m1-1 && p2 = m2 && 1 <= m1 <= 2 && b1 = 20*m1 &&
   max(2, 20*m2+1) <= b2 <= min(20*m2+20, 59) };
```

**Fig. 5.1.** Example maps for communication due to reference `B(i-1,j)` in the Jacobi kernel of Figure 2.1, assuming $\mathtt{N} = 60$

To illustrate how these maps are computed, Figure 5.1 shows some of the intermediate results and the final resulting maps for a single non-local reference $\mathtt{B}(i-1, j)$ in the Jacobi kernel example of Figure 2.1, assuming $N = 60$. (We chose this very simple example to make it easy to understand the maps, although it does not illustrate the generality of our integer set formulation.) Here, the final $SendCommMap(\underline{m})$ specifies that processor **myid** (whose index tuple is $\underline{m} = \{m_1, m_2\}$) must send the boundary values $\mathtt{B}(20m_1 + 20, 20m_2 + 1 : 20m_2 + 20)$ to its right neighbor, $\underline{p} : p_1 = m_1 + 1, p_2 = m_2$, except that processors with $m_1 = 2$ do not send any data. $RecvCommMap(\underline{m})$ specifies that processor **myid** must receive the boundary values $\mathtt{B}(20m_1, 20m_2 + 1 : 20m_2 + 20)$ from its left neighbor, $\underline{p} : p_1 = m_1 - 1, p_2 = m_2$, except processors with $m_1 = 0$ do not receive any data. The *min* and *max* terms in the maps exclude the communication of edge elements that are not accessed. The key steps used to compute these maps are as follows.

For a given reference, we first compute a map, *DataAccessedMap*, describing the entire set of data (local and non-local) accessed by each processor $\underline{p}$ in all iterations of the loops out of which communication has been vectorized. This is done by composing the computation partitioning and reference maps (see Figure 5.1). Then, for a read reference, the set of non-local data accessed by the processor $\underline{m}$ (denoted $nlDataAccessed(\underline{m})$) is the difference between the data accessed and the data owned by that processor. For a write

reference, the set of non-local data accessed is the *intersection* of the data accessed by $\underline{m}$ and the data owned by all other processors $\underline{p}$, since a write must update all owners of the data. (In the absence of replicated data, this step would be equivalent for reads and writes.) Now, the data that $\underline{m}$ must receive from each processor $\underline{p}$ is the intersection of the non-local data accessed by $\underline{m}$ and the data owned by $\underline{p}$. The data that $\underline{m}$ must send to each processor $\underline{p}$ is the intersection of the non-local data accessed by $\underline{p}$ and the data owned by $\underline{m}$. For a single reference, the last two results are exactly RecvCommMap($\underline{m}$) and SendCommMap($\underline{m}$) respectively. To coalesce communication for multiple non-local references (including both reads and writes), we simply take the union of these maps over all coalesced references.

If any of the array elements accessed by a read reference is replicated, a simple additional step is necessary to ensure that only a single owner sends the element to each processor that reads it. Similarly, if the CP for a write reference is replicated, a similar step ensures that only one writer sends the data back to each owner. To avoid communication bottlenecks, we ensure that all the owners (or writers) participate by providing data to different groups of destination processors [1].

For the case of coarse-grain pipelining, we use another additional step to account for the blocking of communication (i.e., the granularity of the pipeline). Specifically, in the range of the above maps, we extend the dimension to be blocked from a single value to a range, using symbolic bounds to represent the block of data communicated in each pipeline stage.

*SendCommMap* and *RecvCommMap* are used by the code generator to synthesize communication. Several alternative communication strategies can be directly supported. To implement pairwise, point-to-point communication we synthesize separate loops to iterate over the domain (processor set) of each map. Note that these loops will enumerate exactly those processors with which processor `myid` must exchange data. The algorithm described in Section 5.2 is then used to determine whether data to be communicated is contiguous in memory, or if this is not statically provable, to emit a logical expression that checks this at run-time. In the latter case, we also synthesize a loop to iterate over the range of *SendCommMap* (i.e., the data to be sent to $\underline{p}$) and use it to copy the data to a buffer. When we check contiguity at runtime, we generate both in-place and buffered communication for a particular communication event. For the receiving end, data can be received in-place if overlap areas are used and the communicated section is contiguous. The receiver uses the same approach to determine whether to receive data in place or into a communication buffer. We currently use the `MPI_Bsend` and `MPI_Brecv` primitives for synchronous point-to-point communication, which guarantee sender-side buffering. This is a simple solution to ensure that deadlock does not occur in cases where all processors have to send and receive data, but it may introduce excess buffer copy operations at the sender. We are currently extending our communication generation to

use asynchronous message-passing primitives which can significantly reduce buffer copying overheads at the sender and receiver.

It is straightforward to extend the above approach to exploit collective communication primitives, in cases where these will be more efficient than point-to-point communication. For example, using a broadcast operation simply requires eliminating the processor loop and using a single call to broadcast the data set to all processors. The methods to determine if data is contiguous and to generate the buffer packing code both remain unchanged. Reductions require separate code generation steps to synthesize code to compute local partial sums within each processor. The data set for a reduction is simply the entire temporary variable (scalar or array) used to hold the local partial sums. Thereafter, the code generation for buffering and communication steps of a reduction are the same as above.

In summary, code generation for message vectorization and coalescing relies on the integer set framework to compute the processor and data sets for communication, and use these to synthesize code for explicit communication. The algorithms are independent of the specific form of the reference, data layout, and computation partitioning involved, and fully support the general computation partitioning model in dHPF. Independent algorithms are used to determine whether buffering is required, and to minimize the costs of accessing buffered non-local data. These are described in the following subsections.

### 5.2 Recognizing in-place communication

Whether compiling HPF for shared-memory or message-passing architectures, avoiding excess data copies can boost performance. Here we describe a technique developed to avoid data copies when generating code for a message passing communication model based on MPI. Common MPI implementations permit data to be sent or received "in-place" (avoiding an explicit data copy) when the address range of the data is contiguous. To increase the likelihood that communication can be performed in-place, we have developed a combined compile-time/run-time algorithm for recognizing contiguous data based on our capability of generating code from integer-sets.

A communication set for an $n$-dimensional Fortran array represents contiguous data if there is some dimension $k, 1 \leq k \leq n$ such that, for the high-order dimensions $1 \leq i < k$, the set spans the full range of array dimension $i$, along dimension $k$ the set has a contiguous index range, and in the low-order dimensions $k+1 \leq j \leq n$, the set contains a single index value. Using our integer-set representation for communication sets, we can express these individual conditions as Boolean predicates, requiring up to three predicates for each array dimension. The existence of a value $k$ satisfying the above properties can be directly expressed as a logical combination of these predicates, first eliminating those predicates that can be proven true or false at compile time. We construct and express this logical expression as

an integer set. If the entire expression can be proven true or false at compile time, then we generate a single version of code for communicating contiguous or non-contiguous data respectively (i.e., with and without buffer packing). If the expression cannot be proven true or false, we synthesize code from the integer set to test the condition at runtime. In this case, we generate both versions of communication code (with and without buffer packing).

Directly checking the logical condition constructed above requires checking $O(n^2)$ terms (conjunctions of predicates), at compile-time or runtime. We can in fact reduce this cost to $O(n)$ by using a single scan of the dimensions (leftmost first) to find the first dimension $k$ which cannot be proved to span the full range of the array dimension, and then checking the predicates for $k \ldots n$. If these predicates cannot be proven at compile time, we can synthesize code to repeat this scan and check at runtime, when it can be done precisely. In practice, however, the number of array dimensions $n$ is typically small, and many of the predicates are statically proven true or false. The cost of evaluating the logical condition at run-time will usually be much smaller than the cost of packing all but the smallest messages into a communication buffer. Therefore, we take the simpler approach (described above) of constructing and evaluating the logical condition directly as a single set.

By combining compile-time and runtime decisions to identify contiguous data, we obtain maximum efficiency when the data is provably contiguous but also minimize the likelihood that explicit buffer packing will be needed when this decision cannot be made until runtime. Furthermore, by basing the analysis directly on an explicit integer set representation of the data, we can apply it to arbitrary communication sets, independent of data layouts and communication patterns.

### 5.3 Implementing Loop-Splitting For Reducing Communication Overhead

As described at the outset of this section, loop-splitting (or iteration reordering) techniques can be used to ameliorate two types of communication overhead: the latency of communication, and the cost of referencing buffered non-local data. Both techniques involve splitting a loop to separate the iterations that access only local data from those that may access non-local data. After splitting, we can overlap communication for the loop with the computation in the local iterations, which do not require the communicated data. We can also reduce the number of ownership guards executed before non-local references by eliminating the guards in the local iterations, and perhaps some in the non-local iterations as well.

The only implementation of loop-splitting we know of is in Kali [32], where the authors used set equations to explain the optimization but used case-based analysis to derive the iteration sets (during compiler development) for a few special cases restricted to one-dimensional data distributions. This

approach is only practical for a small number of special cases. We have extended the equations in [32] to apply to an arbitrary number of non-local references with any regular data layouts and any CP in our CP model, using the sets and mappings described in previous sections. We first describe the loop-splitting analysis for communication overlap, because the loop transformations in this case subsume the transformations required for splitting for buffer access.

Loop-splitting in dHPF is applied one perfectly nested sequence of loops at a time, for loops that satisfy the conditions below. The restriction to perfectly nested loops is not essential, but slightly simplifies our implementation of the code generation. The analysis described here can be used unchanged for imperfect loop nests. We look for a maximal loop-nest that includes loops for which it is legal to reorder the loop iterations, and also loops which we can predict will not to be reordered. It is legal to reorder iterations of a loop if it has no loop-carried data dependences and it does not enclose any communication operations. We can predict that certain loops will not be reordered, as follows. Note that for a *read* reference, a subscript in a non-distributed array dimension will not cause the reference to be non-local. (In a write reference, however, such a subscript could induce communication to send the data back to all the owners.) Therefore, non-local index set splitting will not reorder the iterations of a loop whose loop index variable is used to index only local references and *non-distributed* array dimensions in non-local RHS references. Therefore, we can ignore any dependences that may be carried on such loops, and these loops can be safely included in the loop nest.

Since any write reference as well as any read reference may be non-local in dHPF, our goal is to separate the iterations of a loop nest into four sections: those that access only local data ($localIters$), and those that only read, only write, or read and write non-local data ($nlROIters$, $nlWOIters$ and $nlRWIters$ respectively). These sets are computed using a sequence of integer set equations, taking as input the basic sets and maps of Figure 3.1 and the non-local data set, $nlDataAccessed(\underline{m})$, computed as an intermediate result for the communication sets as described in Section 5.1. The detailed equations are explained in [1]. The key step is computing the iterations that access non-local data for each given reference. The non-local data set for the reference describes the non-local data accessed by the processor `myid`. Composing this map with $RefMap^{-1}$ gives the iterations that access these non-local elements. For example, consider the reference `B(i-1,j)` in the Jacobi kernel of Figure 2.1. We compose $RefMap^{-1} = \{[b_1, b_2] \rightarrow [l_1, l_2] : b_1 = l_1 - 1 \wedge b_2 = l_2\}$ with the non-local data set $nlDataAccessed(\underline{m})$ shown in Figure 5.1. This yields precisely the set of boundary iterations that access non-local data due to that reference: $\{[l_1, l_2] : 1 \leq m_1 \leq 2 \wedge l_1 = 20 * m_1 + 1 \wedge max(2, 20 * m_2 + 1) \leq l_2 \leq min(20 * m_2 + 20, 59)\}$. By taking unions over non-local read and write references respectively, we directly get the set of iterations that read non-local data and the set of iterations that

| | |
|---|---|
| `SEND data for non-local reads` | `SEND data for non-local reads` |
| `execute` **nlWOIters** | `execute` **nlWOIters** |
| `SEND data for non-local writes` | `execute` **localIters** |
| `execute` **localIters** | `RECV data for non-local reads` |
| `RECV data for non-local reads` | `execute` **nlROIters** $\bigcup$ **nlRWIters** |
| `execute` **nlROIters** | `SEND data for non-local writes` |
| `RECV data for non-local writes` | `RECV data for non-local writes` |
| (a) if *nlRWIters* is empty | (b) if *nlRWIters* is non-empty |

**Fig. 5.2.** Generated code for overlapping communication and computation.

write non-local data. These sets may not be disjoint, but from these two sets and the CP, the four desired sets can be directly computed. The code for these individual loop sections is directly synthesized from the respective integer sets by using MMCODEGEN.

We schedule the communication and computation for this loop nest in the sequence shown in Figure 5.2. Both read and write communication latency would be overlapped with some computation if all non-local writes are performed first, then local iterations, and finally non-local reads. This is possible when *nlRWIters* is empty, as shown in Figure 5.2(a). If *nlRWIters* is non-empty, however, these iterations both read and write non-local data, and these must be placed after the RECV for non-local reads and before the SEND for non-local writes. Therefore, we can overlap either read or write communication with *localIters*, but not both. A simple heuristic could be used to choose between the two alternatives, by comparing how early read data is produced and how late write data is consumed. For now, however, we simply overlap read communication with *nlWOIters* and *localIters*, and we merge *nlRWIters* with *nlWOIters*, as shown in Figure 5.2(b).

The goal of the second optimization based on loop-splitting is to minimize the number of ownership guards executed before non-local references. A reference would not need a guard if it accesses only local or only non-local data in all iterations of the enclosing loops. Therefore, in a loop with $r$ non-local references, the ownership guards could be completely eliminated by splitting the index space into $2^r - 1$ non-local subsets, plus the local section. To avoid this exponential behavior, we can simply split the loop into one local and one non-local subset, and use guards only in the non-local section if necessary. If splitting is being applied for communication overlap, that actually creates three non-local sections instead of one, and no further transformations are required. In any case, references in the local iterations do not need ownership guards . A reference in a non-local loop section also does not need such guards if is the set of iterations in that section is identical to the set of non-local iterations due to that reference alone.

```
      parameter (N=64)
      real a(N), b(N), f(N,N,N)
CHPF$ processors p(4)
CHPF$ template t(N,N,N)
CHPF$ distribute t(*,*,block) onto p
CHPF$ align f(i,j,k) with t(i,j,k)
CHPF$ align a(k)     with t(*,*,k)
CHPF$ align b(k)     with t(*,*,k)

  do j=1,N
    do k=2,N
C     SEND f(1:N,j,k-1)                ! ON_HOME f(1:N,j,k-1)
C     RECV f(1:N,j,k-1)                ! ON_HOME f(1:N,j,k)
      do i=1,N
                                       ! ON_HOME f(i,j,k)
        f(i,j,k) = (f(i,j,k) - a(k) * f(i,j,k-1)) * b(k)
      enddo
    enddo
  enddo
```

**Fig. 6.1.** HPF source for a fragment from the Erlebacher benchmark, showing the preliminary placement of `SEND` and `RECV` and the CPs assigned.

Code generation for the loop transformations described above is integrated into the hierarchical code generation framework described in Section 4, as an alternative computation partitioning strategy. This is because code generation from the local and non-local sets has the side-effect of enforcing the CPs assigned to the statements in the loop (including reducing the loop bounds and introducing guards if necessary). This follows because each of the four loop sections is a subset of the iterations assigned to processor `myid` by these CPs. Thus, the combination of the integer-set-based analysis and the hierarchical code generation framework made it quite simple to add even these relatively complex optimizations in the compiler.

# 6. Control flow simplification

## 6.1 Motivation

Several of the strategies for partitioning computation can split a loop's iteration space to avoid adding computation partitioning guards inside the loop. By splitting the iteration space, such guards can be added between sections of the loop instead of inside, which can dramatically reduce the dynamic execution frequency of guards. However, after a loop has been split, the smaller resulting iteration spaces provide a sharper context that may make some conditionals or loops nested inside redundant or unsatisfiable. We illustrate these effects with an example later in this section.

```
     do j = 1, 64

1      if (pmyid1 >= 1)
2        k = 16 * pmyid1 + 1

         !--<< Iterations that access only local values >>--
3        if (16 * pmyid1 <= k - 2)                        !INFEASIBLE
           COMPUTE f(1:64, j, k)

4        if (16 * pmyid1 == k - 1  && pmyid1 >= 1)       !TAUTOLOGY
           RECV f(1:64, j, 16*pmyid1)

         !--<< Iterations that read non-local values >>--
5        if (16 * pmyid1 >= k - 1)                        !TAUTOLOGY
           COMPUTE f(1:64, j, k)
6      do k = 16 * pmyid1 + 2, 16 * pmyid1 + 16
7        if (16 * pmyid1 == k - 17 .and. pmyid1 <= 2)  !INFEASIBLE
           SEND f(1:64, j, k - 1)

         !--<< Iterations that access only local values >>--
8        if (16 * pmyid1 <= k - 2)                        !TAUTOLOGY
           COMPUTE f(1:64, j, k)

9        if (16 * pmyid1 == k - 1 && pmyid1 >= 1)        !INFEASIBLE
           RECV f(1:64, j, 16*pmyid1)

         !--<< Iterations that read non-local values >>--
10       if (16 * pmyid1 >= k - 1)                        !INFEASIBLE
           COMPUTE f(1:64, j, k)
       enddo
       if (pmyid1 <= 2)
         k = 16 * pmyid1 + 17
11       if (16 * pmyid1 == k - 17 .and. pmyid1 <= 2)  !TAUTOLOGY
           SEND f(1:64, j, k - 1)
     enddo
```

**Fig. 6.2.** Skeletal SPMD code for Fig. 6.1 with partitioned computation.

If individual code generation steps attempted to exploit full contextual knowledge to avoid or eliminate guards and empty loops, they would have to be significantly more complex. Furthermore, implementing such an approach would also require rebuilding or incrementally updating analysis information after each code generation step. We use Instead, we use a simpler and less expensive approach in which we make no effort to avoid control-flow complexity that arises as a result of interactions between the different optimization steps, and instead use a separate post-pass optimization to eliminate excess control flow. This approach has two major advantages: existing code generation algorithms can be simpler, and the post-pass control-flow simplification can use powerful algorithms that exploit global information about the program.

```
do j = 1, 64

  if (pmyid1 >= 1) then
    k = 16 * pmyid1 + 1
    RECV f(1:64, j, k-1)
    !--<< Iterations that read non-local values >>--
    COMPUTE f(1:64, j, k)
  endif !(pmyid1 >= 1)

  do k = 16 * p_myid1 + 2, min(16 * p_myid1 + 16, 63)
    !--<< Iterations that access only local values >>--
    COMPUTE f(1:64, j, k)
  enddo !k

  if (p_myid1 <= 2) then
    k = 16 * p_myid1 + 17
    SEND f(1:64, j, k - 1)

enddo !j
```

**Fig. 6.3.** Skeletal SPMD code for Fig. 6.2 after simplification.

The foundation for control flow simplification in dHPF is an algorithm for globally propagating symbolic constraints on the values of variables imposed by loops, conditional branches, assertions, and integer computations. Several previous systems have supported strategies for computing and exploiting range information about variables [20, 8, 9, 25, 44]. Two key differences that distinguish our work are that we handle more general logical combinations of constraints on variables (not just ranges) and we use these constraints to simplify control flow. In this section, we present an example that illustrates how a sequence of code generation steps results in superfluous loops and conditionals and then provide a brief overview of our control-flow simplification technique. Our algorithm is described and evaluated in detail in [34].

To illustrate some of the principal sources of excess control-flow in dHPF, Fig. 6.1 shows source code for a loop from the Erlebacher benchmark, including the CPs and the initial placement of communication chosen by the compiler. The assignment statement is given the CP ON_HOME `f(i,j,k)`, and communication for the non-local reference `f(i,j,k-1)` is placed inside the `k` loop because of a loop-carried flow dependence for array `f`. Fig. 6.2 shows the skeletal SPMD code after CP code generation (the entire `i` loop has been shown as "`COMPUTE f(1:64,j,k)`" to simplify the figure). Examination of the code in Fig. 6.2 shows that many of the guard expressions are infeasible or tautological.

The causes of excess control flow in this example are as follows:

1. The guards on lines 8 and 10 are initially generated by splitting the `i` loop into local and non-local iterations. At this time it is not known what

sections of the `k` loop will finally be generated. We cannot apply non-local index set splitting to the $k$ loop because of the loop-carried dependence. However, when MMCODEGEN is applied to reduce loop bounds for the `k` loop, the loop is tiled into 3 sections ($\mathtt{k} = 16 * \mathtt{pmyid1} + 1$, $\{16 * \mathtt{pmyid1} + 2 \leq \mathtt{k} \leq 16 * \mathtt{pmyid1} + 16\}$, and $\mathtt{k} = 16 * \mathtt{pmyid1} + 17$), so that additional guards are not required within the loop to enforce the different CPs of the three inner statements. The guards on lines 8 and 10 now get duplicated on lines 3 and 5, along with their enclosed **COMPUTE** blocks. Now, in the refined context of these `k` loop sections, the guards on lines 3 and 10 are always false, and those on lines 5 and 8 are always true.

2. When the `k` loop is split as described above, the **SEND** and **RECV** placeholders are duplicated (as shown) during CP code generation because of the CPs assigned to the communication placeholders. The CPs assigned to placeholders, shown in Figure 6.1, simply specify that the owner of `f(1:N,j,k)` must receive data (since it will execute the `i` loop), and the owner of `f(1:N,j,k-1)` must send data (since it owns the data being read). These CPs are imprecise in that the **SEND** and **RECV** should actually execute only in the boundary iterations. Precise CPs for communication are difficult to compute and express in any general manner since communication patterns can be quite complex. Instead, we rely on the communication generation phase to insert guards to ensure that only required communication occurs. Those guards are the ones shown on lines 4, 7, 9 and 11. Although the loop context created when partitioning the `k` loop makes these guards infeasible or redundant, the communication code is generated without precise knowledge of this refined loop context.

It is important to note that the excess guards that arise in the code generation steps described above are not due to poor code generation algorithms. In the first case, they result from a sophisticated loop transformation that reduces the dynamic execution frequency of guards. In the second case, avoiding eliminating redundant guards when instantiating communication would require full knowledge of sharper context created by earlier code generation steps.

Another source of excess control flow not illustrated by the example above is the insertion of ownership guards for non-local references. As explained in Section 5.3, we create one to three sections of non-local iterations, and insert ownership guards in each of the non-local sections. However, if the iteration set of any non-local section is non-convex, MMCODEGEN automatically tiles the section into convex regions, potentially making some or all of the guards in that section redundant. In each case, it is sensible to generate these guards oblivious of their context and let a later control-flow simplification pass eliminate any that became unnecessary. This enables our guard insertion algorithms to remain relatively simple without hurting performance. In the next section we describe our global algorithm for eliminating superfluous control flow.

### 6.2 Overview of Algorithm

The algorithm for simplifying control flow is based on the property that each conditional branch node (i.e., `IF` or `DO` loop) in a program guarantees certain constraints on the values of variables for the statements control dependent on the branch node. The goal of our algorithm is to collect and propagate these constraints globally through the code and use this information to simplify the control flow. Our algorithm combines three key program analysis technologies, namely, the control dependence graph (CDG) [13], global value numbering based on thinned gated single assignment form [21], and simplification of integer constraints specified as Presburger formulae [27]. The former two enable us to derive an efficient, non-iterative algorithm for propagating constraints along control dependences, while the latter enables simplification of logical constraints and code generation from the simplified constraints. The information we collect is closely related to the concept of "iterated control dependence" path conditions described by Tu and Padua [44].

Rather than computing constraints on variables directly, we compute constraints on value numbers representing unique values of variables. This allows us to avoid invalidating constraints after redefinitions and SSA merge points since each of these points simply yields a new value number for the variable. Constraints on an old value number may be irrelevant but are never incorrect. Constraints on a value number $V$ at a given statement $S$ in the program are logical combinations of equalities and inequalities that hold true for $V$ at $S$. We use integer sets of rank 0 to represent constraints; this enables us to exploit the Omega library's capabilities for symbolic simplification and code generation.

Briefly, our constraint simplification algorithm operates in two passes as follows. The first pass collects constraints at conditional branch nodes in a single reverse-post-order traversal of CDG. This order ensures that all control dependence (CD) predecessors of a node are visited before the node itself, except for predecessors along back edges which form cycles in the CDG. At each point in the traversal, we collect the constraints for a node as the intersection of the constraints enforced locally at the node with the disjunction of the constraints that hold along paths from each of its control dependence predecessors. Our single pass algorithm for collecting constraints computes a conservative approximation in that it ignores constraints along back edges.[4] Despite using a non-iterative strategy for collecting constraints, we are still often able to extract useful constraints for loop-variant iterative values, particularly auxiliary induction variables recognized when computing value numbers. This includes relatively complex auxiliary induction variables that do not have a closed form but are defined by a loop-invariant iterative function (e.g., $i = i * 2$).

[4] It is safe to ignore these constraints because our use of value numbers ensures that constraints along back edges will never invalidate existing constraints, rather they would just add more information.

The second pass makes a reverse preorder traversal of the CDG (visiting CD dependents before ancestors), using the computed constraints to simplify a procedure's control flow. The bottom-up order we choose to simplify constraints is a convenience that simplifies bookkeeping by ensuring that code transformations we apply at a node will not eliminate any conditionals that we will subsequently reach later in our iteration. If the outgoing constraints for a loop or conditional branch are unsatisfiable, its code is eliminated. If the outgoing constraints at a conditional branch are implied by its incoming constraints, the test is eliminated. Otherwise, the test may still be simplified given the known information in the incoming constraints. In this case, the simplified constraints are used to regenerate a simpler guard using the MMCODEGEN operation.

The algorithm also takes into account *assertions* specifying "known" constraints about program variables inserted into the code by the programmer or previous phases of the compiler. These assertions provide a mechanism for communicating information that a later compiler pass may be unable to infer directly from the code. We refer the reader to [34] for the details of the overall algorithm, including the handling of iterative constructs and assertions.

### 6.3 Evaluation and Discussion

Figure 6.3 shows the simplified code for the Erlebacher intermediate code shown in Figure 6.2. In Figure 6.3, we see that all infeasible and tautological guards have been eliminated, the latter being replaced by their enclosed code. We also evaluated the algorithm for 3 benchmarks (Tomcatv from the Spec92 benchmark suite, Erlebacher, and Jacobi) [34]. The algorithm eliminated between 31% and 81% of guards introduced by the compiler for these programs, yielding between 1% and 15% improvement in execution time on an IBM SP-2. These improvement are achieved over and above the many aggressive optimizations in dHPF and scalar optimizations performed by the SP-2 node compiler. Overall, the control-flow simplification is able to compensate for the lack of complete context information in the earlier code generation phases (and for the overly simple CPs for communication statements), permitting the earlier phases to be simpler without impacting performance.

An interesting outcome we observed in our experiments is that the general purpose control-flow simplification algorithm (in combination with our CP code generation algorithm and loop splitting) provides some or all of the benefits of much more specialized optimizations such as vector-message pipelining [43] and overlap areas [14]. In particular, the pipelined shift communication pattern for Erlebacher shown in Fig. 6.3 is exactly what vector message-pipelining aims to produce, but the latter is a narrow optimization and complex to implement, as explained in [34]. In dHPF, however, it results naturally from loop-splitting, bounds reduction, and control-flow simplification. As a second example, overlap areas are specifically designed to simplify

referencing non-local data in shift communication patterns, but a comprehensive implementation of overlap areas requires interprocedural analysis. Even without overlap areas, we obtained equally simple code for shift communication patterns in both Tomcatv and Erlebacher, and nearly as simple code in Jacobi. This happened because non-local loop-splitting and control-flow simplification together eliminated all or most of the ownership guards on non-local references, so that it was equally simple to access data out of separate non-local buffers as out of overlap areas. In fact, the code for Tomcatv without overlap areas slightly outperformed the code with overlap areas (by about 3%) because overlap areas required an extra unpacking operation [34]. Overall, these results and the evaluation described above indicate that the control-flow simplification algorithm can be a useful general-purpose optimization for parallelizing compilers.

## 7. Conclusions

The motivation for the development of the Rice dHPF compiler is the need for HPF compiler technology that approximates or exceeds hand-coded performance across a broad spectrum of programs. Meeting our goal of offering consistently high performance will require a large collection of optimizations that are uniformly applicable to a wide variety of programs.

In this chapter, we described several static code generation techniques that enable aggressive and robust optimizations in an HPF compiler, and yet greatly simplify the implementation of many of these optimizations. We described a computation partitioning model, a framework for analysis and optimization, and a code generation strategy that are more general than those used previously by HPF compilers. The key conclusions we draw from this work are as follows. First, a uniform and comprehensive code generation framework can support a very general computation partitioning model, and permit harnessing powerful code-generation algorithms that extract the full performance of partitionings enabled by the model. Second, the use of abstract set equations as the medium for expressing optimizations has greatly simplified the construction of even complex optimizations, and therefore makes it practical to incorporate a comprehensive collection of optimizations in a compiler. In addition, the high-level nature of these equations makes the optimizations very broadly applicable (including any computation partitionings in the general CP model), and increasing their overall impact. Third, a global control-flow simplification algorithm can substantially reduce excess control-flow in the generated code, permitting other code generation algorithms to be significantly simpler. Finally, the synergy between some of these general optimizations provides some or all of the benefits of much more specialized optimizations such as vector-message pipelining and overlap areas.

The base technology (the Omega library) used to implement the integer set framework is experimental, but it has proved invaluable for prototyping

optimizations based on the framework. Further experience from using dHPF on a wide variety of programs will be required to judge whether the technology is efficient enough to be practical for commercial implementations. If this approach proves too costly, it would be important to develop more efficient but less precise set representations so that compilers could still obtain the power and simplicity of the integer set formulations.

In this chapter, we have focused on the problem of compiling "regular" data parallel codes. Many applications have features that cause them to fall outside this class, and in such cases different compilation strategies and run-time support are more appropriate. Our hierarchical framework for code generation enables different code generation strategies to be applied to a scope, or in some circumstances even to a single statement. This capability provides the flexibility needed to integrated alternate methods such as inspector/executor or run-time resolution to cope with programs that are not statically analyzable.

Although the analysis and optimization described here focused on compiling for a message-passing target machine, most if not all of this technology is directly applicable to uniform and distributed shared memory (i.e., SMP and DSM) systems. Some of the optimizations built on the program analysis framework are aimed at avoiding communication by maximizing data access locality, which is essential for DSM systems as well. Furthermore, for shared-memory systems built from commodity microprocessors, managing locality by managing the memory hierarchy is essential for performance. The analysis and code generation capabilities described here are guided primarily by array layouts and provide the right leverage for exploiting locality and optimizing for the memory hierarchy on this emerging class of systems.

## Acknowledgements

The dHPF compiler is the result of the collective efforts of all the members of the dHPF compiler group. Ken Kennedy provided invaluable guidance and leadership that have made the overall project feasible. Ajay Sethi contributed to the design and implementation of the set-based formulation of communication. Lisa Thomas implemented the algorithm to detect in-place communication from integer sets. Bill Pugh and Evan Rosser provided valuable support and advice on using the Omega library. The Omega calculator, a scripting interface to the Omega library, proved extremely useful for experimenting with integer set equations for our compiler. Sarita Adve, Robert Fowler, Ken Kennedy and Evan Rosser provided valuable comments on earlier drafts of this paper.

This work has been supported in part by DARPA Contract DABT63-92-C-0038, the Texas Advanced Technology Program Grant TATP 003604-017, and sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number

F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency and Rome Laboratory or the U.S. Government.

## References

1. V. Adve, J. Mellor-Crummey, and A. Sethi. HPF analysis and code generation using integer sets. Technical Report CS-TR97-275, Dept. of Computer Science, Rice University, April 1997.
2. S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.
3. C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.
4. C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.
5. P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.
6. R. Barua, D. Kranz, and A. Agarwal. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.
7. S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.
8. W. Blume and R. Eigenmann. Demand-driven symbolic range propagation. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, OH, August 1995.
9. François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, June 1993.
10. Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 704–709, San Francisco, CA, February 1995.
11. S. Chakrabarti, M. Gupta, and J-D. Choi. Global communication analysis and optimization. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
12. S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, Xerox Corporation, December 1992.

13. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
14. M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.
15. M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication for multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.
16. M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.
17. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, February 1996.
18. R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems*. PhD thesis, Dept. of Computer Science, Rice University, December 1994.
19. J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal of Digital Equipment Corp.*, 7(3):5–23, Fall 1995.
20. W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.
21. Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.
22. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
23. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.
24. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.
25. Harold Johnson. Data flow analysis of 'intractable' imbedded system software. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 109–117, 1986.
26. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.
27. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, April 1996.
28. K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
29. Ken Kennedy and Charles Koelbel. *The High Performance Fortran 2.0 Language*, chapter 1. Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

30. Ken Kennedy and Ajay Sethi. Resource-based communication placement analysis. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.
31. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
32. C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.
33. J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.
34. J. Mellor-Crummey and V. Adve. Simplifying control flow in compiler-generated parallel code. Technical Report CS-TR97-278, Dept. of Computer Science, Rice University, May 1997.
35. S. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, August 1995.
36. D. Oppen. A $2^{2^{s^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, July 1978.
37. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
38. J. Ramanujam. *Integer Lattice Based Method for Local Address Generation for Block-Cyclic Distributions*, chapter 17. Lecture Notes in Computer Science Series. Springer-Verlag, 1997.
39. A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.
40. J. Saltz. *Runtime Support for Irregular Problems*, chapter 17. Lecture Notes in Computer Science Series. Springer-Verlag, 1997.
41. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.
42. J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.
43. C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.
44. Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.
45. Kees van Reeuwijk, Will Denissen, Henk Sips, and Edwin Paalvast. An implementation framework for hpf distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):897–914, September 1996.
46. H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

$$I_1 = \{[i,j] : 1 \le i \le 10 \ \wedge \ 1 \le j \le 5 \wedge \ c \ge 1\}$$

$$I_2 = \{[i,j] : 1 \le i \le 6 \ \wedge \ 1 \le j \le 5 \ \wedge \ c \ge 4\}$$

$$known = \{c \ge 1\}$$

(a) Two iteration spaces.

```
do i = 1, 10
  if (c .ge. 4 .and. i .le. 6)
    do j = 1, 5
      s1(i,j)
      s2(i,j)
  if (i .ge. 7 .and. c .ge. 4)
    do j = 1, 5
      s1(i,j)
  if (c .le. 3)
    do j = 1, 5
      s1(i,j)
enddo  !i
```

(b) Code template from MMCODEGEN.

**Fig. 7.1.** Constructing a code template from iteration spaces.

## Appendix A. MMCODEGEN

Here we briefly introduce Kelly, Pugh & Rosser's algorithm for "multiple-mappings code generation" [26], which we refer to as MMCODEGEN. This algorithm is available as part of the Omega library, and serves as a cornerstone for a variety of code generation tasks in the dHPF compiler. The inputs to MMCODEGEN are as follows:

MMCODEGEN($I_1 \ldots I_v$, `known`, `effort`) :

| | | |
|---|---|---|
| $I_1 \ldots I_v$ | : | Iteration spaces for $v$ statements. |
| `known` | : | A set of rank 0, giving constraints on global variables in $I_1 \ldots I_v$ that will be externally enforced. |
| `effort` | : | Integer specifying to remove conditionals `effort` + 1 inner loops |

From these inputs, MMCODEGEN synthesizes a code template that enumerates the tuples in $I_1 \ldots I_v$ in lexicographic order, "$\prec$", where the same tuple in different sets is ordered as: $(\underline{i} \in I_j) \prec (\underline{i} \in I_k), j < k$. The constraints in *known* are assumed true and will not be enforced within the code template. The innermost `effort` + 1 loops in the code sequence will not contain conditionals. Figure 7.1 shows two iteration spaces and the corresponding code template generated with $effort = 0$, i.e., one level of guard lifting. In the code template, `s1` and `s2` are placeholders representing the first and second statements, respectively. The digit suffix in the placeholder name represents the *index* of the iteration space it represents.

When MMCODEGEN is applied to a set of rank zero, it degenerates to synthesizing an IF statement that tests the constraints of the set. This variant arises (for example) for generating guards to enforce CPs for statements outside loops, for control-flow simplification, and for code generation for testing at runtime whether communication can be performed in place.