An Integer Set Framework for HPF Analysis and Code Generation

Vikram Adve John Mellor-Crummey Ajay Sethi

Department of Computer Science, M/S 132
Rice University
6100 Main Street, Houston, TX 77005-1892.

Abstract

The core of the Rice dHPF compiler for High Performance Fortran is a practical, executable, equational framework for data parallel program analysis and optimization. This framework, based on abstract operations on sets of integers, greatly simplifies the implementation of many key analysis, optimization and code generation tasks (even with a general computation partitioning model), and yet enhances their generality and flexibility. This paper describes the equational framework, tradeoffs in its implementation, the formulation and implementation of important analyses and optimizations within the framework, and a hierarchical code generation strategy to support the framework.

1 Introduction

Data-parallel languages like High-Performance Fortran (HPF)[21] hold out the promise of a high-level, portable, parallel programming model usable by non-computer-scientists. To achieve wide acceptance, such a language will require parallelizing compilers that can extract consistently high performance from a wide variety of applications. The Rice dHPF compiler project is developing a range of optimization techniques that can help deliver such consistent levels of performance.

To achieve these levels of performance, we believe two requirements must be met. First, the compiler should have the maximum possible flexibility in its computation partitioning strategies, and not restrict the partitioning to follow the widely-used owner-computes rule [25] or restrict all statements in a loop to have the same partitioning [1]. The dHPF compiler incorporates a flexible computation partitioning (CP) model, in which each statement (including control flow) has one or more computational "homes" that specify where instances of the statement will execute. Second, the compiler must provide a wide variety of optimization techniques and apply these with as few restrictions as possible on the form of references, data distributions, and computation partitionings.

Program analysis and implementation techniques in current compilers, however, do not appear flexible enough to support a general class of computation partitionings and program optimizations. Most research and commercial data-parallel compilers to date [23, 6, 22, 28, 17, 10, 11, 13, 7, 12, 31] (including the Rice Fortran D compiler) perform communication analysis and code generation using pattern matching. While such approaches can provide excellent performance where they apply, they may provide poor performance for patterns they cannot handle. More importantly, pattern-based compilers require a relatively high development cost for each new optimization because the analysis and code generation for each case must be expressed in terms of specific patterns; this makes it difficult to achieve wide coverage with optimizations to offer consistently high performance.

A few research groups have used systems of linear inequalities to provide a more general and flexible approach than patterns for compiling data parallel programs [3, 2, 5, 1]. In most cases, these have been used primarily for code generation based on Fourier-Motzkin elimination (FME) [26, 3]. Amarasinghe and Lam [1] also describe how inequalities and FME can support array dataflow analysis and a few specific communication optimizations, but these primarily operate by direct manipulation of inequalities. Furthermore, their representation could not represent general non-convex sets or a general set union operation, so optimizations using them are limited in scope [1, 5, 2]. (For example, their representation would not support our CP model, or optimizations such as coalescing of arbitrary affine references to an array [1].)

The Rice dHPF compiler is based on a practical, executable, equational framework that enables data parallel program analyses and optimizations to be expressed in terms of abstract operations on integer sets. Early work on compiling data parallel programs describes basic compilation steps in terms of set operations [16, 22]; however, this was viewed only as a pedagogical abstraction and the corresponding compilers (Kali and Fortran D) were implemented using pattern matching techniques. Using our equational framework, we have devised and implemented simple, yet general, equational formulations of the major partitioning and communication analyses as well as a number of important optimizations. All of the analyses and optimizations within this framework fully support our general computation partitioning model, permit references with arbitrary affine subscripts, and are independent of the specific data layouts. The framework and the optimizations have been implemented in the Rice dHPF compiler, which currently compiles HPF programs to message-passing Fortran with MPI.

This paper describes the equational framework, tradeoffs that arise in the implementation of the framework and how they were addressed in dHPF, and the formulation and implementation of important analyses and optimizations within the framework. Optimizations we have formulated as integer-set equations address message overhead, message latency, the costs of non-local storage management, and scalar efficiency of the generated code. These optimizations include:

- message vectorization for arbitrary regular communication patterns;
- message coalescing for arbitrary affine references to an array;
- loop-splitting transformations to overlap communication with computation *within* a single loop-nest, and also to reduce the overhead of accessing buffered non-local data;
- a combined compile-time/run-time technique to reduce explicit data copies for a message; and
- constraint propagation to simplify code using constraints from enclosing control-flow.

We describe a hierarchical code generation strategy that supports the optimizations and our general CP model. Finally, we present some preliminary benchmark measurements that provide a baseline for the performance of dHPF's generated code.

In the following section, we describe our integer-set based equational framework and the formulation of optimizations using integer sets. In Section 4, we describe our code generation strategy and the use of constraint propagation to simplify the generated code. Finally, we present benchmark measurements, contrast our work with previous work, and then offer some concluding remarks.

2 The Integer Set Framework and its Applications

The following subsection explains the framework of primitive sets and mappings that underlies integer-set-based functions in the dHPF compiler. We then describe how key analyses and optimizations are implemented in terms of these abstractions.

2.1 An Integer Set Framework for Data-Parallel Compilation

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 ([1, 16, 22]). These are:[1]

$data_k$:	the index set of an array of rank $k, k \geq 0$
$loop_k$:	the iteration space of a loop nest of depth $k, k \geq 0$
$proc_k$:	the processor index space in a processor array of rank $k, k \geq 1$
$Layout$:	$proc_n \rightarrow data_k : \{[\underline{p}] \rightarrow [\underline{a}]$: array element $\underline{a} \in data_k$ is allocated to processor $\underline{p} \in proc_n\}$
Ref :	$loop_k \rightarrow data_n : \{[\underline{i}] \rightarrow [\underline{a}]$: array element $\underline{a} \in data_k$ is referenced in iteration $\underline{i} \in loop_k\}$
$CPMap$:	$proc_n \rightarrow loop_k : \{[\underline{p}] \rightarrow [\underline{i}]$: statement instance $\underline{i} \in loop_k$ is assigned to processor $\underline{p} \in proc_n\}$

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" are taken to refer to scalars and outermost statements as well. Note that any mapping we require, including a mapping with domain of rank 0, will be invertible.

We formulate many of the key compiler analyses and optimizations in dHPF directly as abstract set operations on these sets and mappings. This requires an integer set package that supports all the key set operations including intersection, union, difference, domain, range, composition, and projection. (Appendix A defines some of these operations). Pugh et al. at the University of Maryland have recently developed algorithms for integer set problems represented by Presburger formulas [24]. They use advanced Fourier elimination techniques to provide two key capabilities: a general class of integer set operations including set union and non-convex sets, and an algorithm to generate efficient code for multiple iteration spaces [?]. These algorithms are implemented in an integer set package, the Omega library. From the viewpoint of HPF compilation, a significant drawback of their techniques is that they provide only limited handling of symbolic quantities. We used the Omega library to implement the integer set framework in dHPF, but extended the framework to circumvent these limitations. In Section 3, we discuss these tradeoffs and how they were addressed in our implementation.

In practice, the sets $Loop$ and $Proc$ and the mappings $Layout$ and Ref are constructed directly from our IR, and form the primary inputs for further analyses. Their construction takes advantage of a powerful symbolic representation in our compiler, 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, and their construction subsumes simple expression simplification, constant propagation, auxiliary induction variable recognition, and range information for expressions of loop index variables. A value number can be reconstituted back into an equivalent Abstract Syntax Tree (AST) representation of the expression.

Figure 1 illustrates simple examples of the primitive sets and mappings for an example HPF code fragment. The construction of the Layout mapping follows the two steps used to describe an array layout in HPF, 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) [21]. The ON_HOME CP notation and construction of $CPMap$ are described in the following section.

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

[2] The set $\{[] : cond\}$ should be interpreted as a boolean that takes the values true or false, depending on whether the condition $cond$ (expressed as constraints on global variables) is satisfied.

```
real A(0:99), 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)

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

symbolic N

$$
\begin{aligned}
Align_A &= \{[i,j] \rightarrow [i',j'] : i' = i+1 \wedge j' = j\}\\
Align_B &= \{[i,j] \rightarrow [i',j'] : j' = i\}\\
Dist_T &= \{[i,j] \rightarrow [p] : 25p+1 \leq j \leq 25(p+1) \wedge 0 \leq p \leq 3\}\\
Layout_A &= Align_A^{-1}(Dist_T^{-1})\\
&= \{[p] \rightarrow [i,j] : max(25p,1) \leq j \leq 25p+24 \wedge 1 \leq i \leq 100 \wedge p \leq 3\}\\
Layout_B &= Align_B^{-1}(Dist_T^{-1})\\
&= \{[p] \rightarrow [i,j] : max(25p+1,1) \leq i \leq min(25p+25,100) \wedge 1 \leq j \leq 100\}\\
loop &= \{[l_1,l_2] : 1 \leq l_1 \leq N \wedge 1 \leq l_2 \leq N\}\\
CPRef &= \{[l1,l2] \rightarrow [i',j'] : j' = l1 \wedge i' = l2-1\}\\
CPMap &= CPRef^{-1}(Layout_B) \textstyle\bigcap_{range} Loop\\
&= \{[p] \rightarrow [l_1,l_2] : 1 \leq l_1 \leq min(N,100) \wedge 2 \leq l_2 \leq min(N,102)\\
&\qquad \wedge 0 \leq p \leq 3\}
\end{aligned}
$$

Figure 1: Construction of primitive sets and mappings for an Example program

2.2 Computation Partitioning Model

A major goal of dHPF is to support a general class of computation partitionings that enable high performance. This requires that all the analysis and code-generation phases in the compiler fully support any partitioning in this class. This would be impractical with a pattern-matching approach; the abstraction of an integer set framework has proved essential for making such analysis and code generation capabilities practical.

The widely-used *owner-computes* rule specifies that a computation happens on the owner (i.e., the "home") of the left-hand-side reference. In dHPF, we generalize this rule and permit the computation partitioning (CP) to be the owner of one or more arbitrary data references. Consider a statement enclosed in a loop nest with iteration space $\underline{i}$; the CP of the statement is specified by a union of one or more ON_HOME terms: $CP : \cup_{k=1}^{k=n}$ON_HOME$\{A_k(f_k(\underline{i}))\}$. An individual term ON_HOME$\{A_k(f_k(\underline{i}))\}$, specifies that the dynamic instance of the statement 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 $f_k(\underline{i})$. To support analyses and code generation for such CPs, for each statement we compute $CPMap$, an *explicit* integer tuple mapping from processors to iteration spaces. Construction of $CPMap$ requires the expression for each index position in $f_k(\underline{i})$ to 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 $CPMap$ to represent the computation partitioning for a statement as follows. Let $Layout_{Ak}$, Ref_k, and $Loop$ represent the layout of array A_k, the index expression $f_k(\underline{i})$, and the enclosing loop iteration space, respectively. The mapping for a single term ON_HOME$\{A_k(f_k(\underline{i}))\}$ is a simple composition of the layout and reference mappings, restricted in range to the loop index space: $CPMap_k = (Layout_{Ak} \circ Ref_k^{-1}) \bigcap_{range} Loop$. The mapping for the full CP is the union of individual mappings:

$$CPMap = \cup_{k=1}^{k=n}(Layout_{Ak} \circ Ref_k^{-1}) \textstyle\bigcap_{range} Loop. \tag{1}$$

[3] In the presence of dynamic REALIGN and REDISTRIBUTE directives, the formulations and analysis using integer sets require no change as long as only a single known layout is possible for each reference in the program. Multiple reaching layouts would require cloning procedures or generating multi-version code to satisfy this condition.

This mapping specifies the processor assignment for a single statement instance in loop iteration $\underline{i}$. The mapping can be vectorized over a range of iterations L of one or more enclosing loops to represent the combined processor assignment for the set of statement instances in those loop iterations, denoted $Vectorize(CPMap, L)$. These directly form inputs to several of the other analyses, described in the following sections.

2.3 Implementing explicit communication

For message-passing systems, data-parallel compilers must compute the set of data that needs to be exchanged between processors and generate efficient code to pack, communicate, unpack, and reference the non-local data. Two key communication optimizations on such systems are *message vectorization* and *message coalescing*. Vectorization attempts to hoist communication for a reference out of one or more enclosing loops in order to send a few large messages instead of many small messages between each pair of processors. To synthesize vectorized communication, the compiler must compute the set of data to send between each pair of processors; these communication sets depend on the reference, layout, and computation partitioning. Message coalescing combines messages for multiple references to eliminate redundant communication and to further reduce the number of messages. Coalescing requires merging the above communication sets (i.e., set union).

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. Later, we generate code from these sets directly. This abstract formulation greatly simplifies the core of the dHPF compiler and enables efficient handling of general classes of computation partitionings and affine references.

Early phases in dHPF compute which references are "non-local" (i.e., might access non-local data), where to place communication for each reference, and which sets of references can have their communication coalesced. We currently assume a model of communication where data computed on a processor other than the owner is first communicated to the owner, and read references obtain off-processor data only from the owner. Therefore, a write reference is "non-local" if the location is owned by one or more processors besides the processor executing the write. A read reference is non-local if the location is not owned by the processor executing the read. We refer to the entire sequence of messages required for a set of coalesced references as a single logical communication event.

Given the sets of coalesced references and the placement of communication, we compute the communication sets for each logical communication event using the inputs and set equations shown in Figure 2. Due to constraints in supporting multiple processor arrays (described below), we compute two separate maps, $SendCommMap(m)$ and $RecvCommMap(m)$ for a fixed representative processor m (or `myid`). The maps respectively specify the data that processor m must send to each other processor $\underline{p}$ and the data that processor m must receive from each other processor $\underline{p}$.

The key operations are as follows. Steps 1 and 2 compute the map $DataAccessed_r$ which specifies the entire set of data (local and non-local) accessed by each processor $\underline{p}$, via reference r in all iterations of the loops out of which communication has been vectorized. Then, the *non-local* data referenced depends on $DataAccessed_r$ and the local data owned by each processor, $Layout_r$ (in our model, it is the difference of these for a read, but the intersection for a write). These operations cannot be performed directly because the domains of these two maps might represent different processor arrays. Instead (step 3), we convert the maps to sets of data for the fixed processor `myid` (represented by the singleton set $\{m\}$) and perform the difference or intersection on sets in the data domain.

We then compute two maps describing the local and non-local data (w.r.t. to the fixed processor m) that must be communicated with each other processor (steps 4,5). The non-local data map ($NLCommMap_r(m)$) specifies data referenced by m to be communicated to or from each processor $\underline{p}$ that owns the data. This map is the restriction of the range of the $Layout_r$ map to the non-local data set, $nlDataSet_r(m)$. The local

Inputs:

$$
\begin{array}{rcl}
ReadRefs, WriteRefs & : & \text{the coalesced read and write references} \\
Ref_r & : & \text{map representing reference } r,\ \forall r \in ReadRefs \cup WriteRefs \\
CPMap_r & : & \text{computation partitioning map for reference } r \\
Layout_{Ar} & : & \text{layout of the referenced array at reference } r \\
V_{comm} & : & \text{loop-level of innermost loop enclosing communication for } r \text{ after vectorization}
\end{array}
$$

Algorithm:

$$
\begin{array}{lrcl}
(1) & CPMap_r^V & = & VectorizeRangeVars(CPMap_r, V_{comm}) \quad \text{(see Appendix A.)} \\
(2) & DataAccessed_r & = & CPMap_r^V \circ Ref_r \\
(3) & nlDataSet_r(m) & = & \big\{ [\underline{a}] : \text{off-procesor array elements referenced by processor } m \big\} \\
 & & = & \begin{cases} DataAccessed_r(\{m\}) - Layout_{Ar}(\{m\}) & \text{if } r \in ReadRefs \\ DataAccessed_r(\{m\}) \cap Layout_{Ar}(\neg\{m\}) & \text{if } r \in WriteRefs \end{cases} \\
(4) & NLCommMap_r(m) & = & \big\{ [\underline{p}] \to [\underline{a}] : \text{off-processor elements referenced by proc. } m \text{ and owned by proc. } \underline{p} \big\} \\
 & & = & Layout_{Ar} \bigcap_{range} nlDataSet_r(m) \\
(5) & LocalCommMap_r(m) & = & \big\{ [\underline{p}] \to [\underline{a}] : \text{array elements owned by proc. } m \text{ to be communicated with proc. } \underline{p} \big\} \\
 & & = & DataAccessed_r \bigcap_{range} Layout_{Ar}(\{m\}) \\
(6) & SendCommMap(m) & = & \bigcup_{r \in ReadRefs} LocalCommMap_r(m) \ \bigcup \ \bigcup_{r \in WriteRefs} NLCommMap_r(m) \\
(7) & RecvCommMap(m) & = & \bigcup_{r \in ReadRefs} NLCommMap_r(m) \ \bigcup \ \bigcup_{r \in WriteRefs} LocalCommMap_r(m)
\end{array}
\tag{2}
$$

Figure 2: Equations for computing communication sets

data map ($LocalCommMap_r$) specifies data owned by m to be communicated to each other processor $\underline{p}$ that references the data. This map is the restriction of the range of the $DataAccessed_r$ map to the local section owned by m.

Finally, we use $LocalCommMap$ and $NLCommMap$ to compute the data to send to and receive from each processor (steps 6,7). $NLCommMap_r(m)$ specifies the data to be sent by m if r is a write, and received if r is a read. $LocalCommMap_r(m)$ gives the data to be received by m if r is a write and sent if r is a read. [4]

$SendCommMap$ 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) and the range (data set) of each map. (The data loop is only required if data must be packed to or unpacked from a buffer. The following subsection describes a general algorithm to determine if these operations are necessary.) This general approach provides efficient communication for arbitrary static communication patterns, and is even sufficient to obtain high performance on important special cases such as shift and transpose communication patterns. It is straightforward to extend this to exploit special communication primitives such as broadcast, where point-to-point communication would be less efficient.

[4] Note that for a replicated array, $LocalCommMap_r(m)$ as computed in step 5 specifies that all processors $m \neq \underline{p}$ should provide data to $\underline{p}$ on a non-local read. We can eliminate the extra communication during code-generation by choosing a single owner to perform each send, e.g., by choosing the particular processor m whose index is equal to $p_i \in \underline{p}$ in each dimension i along which the array is replicated.

2.4 Recognizing in-place communication

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 develop a combined compile-time/run-time algorithm for recognizing contiguous data based on our capability of generating code from integer-sets.

FORTRAN arrays are stored in column-major order. Accordingly, a communication set C for data in an n-dimensional array `A` represents contiguous data if there is some dimension k 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. Let A represent the index set of the array, and define $S_{<i>}$ to be the projection (i.e., range) of set S in dimension i, $1 \leq i \leq \mathrm{rank}(S)$. Then the above condition can be formalized as:

$$\exists k \ s.t. \ 1 \leq k \leq n \quad \wedge \ \bigwedge_{i=1}^{i=k} (C_{<i>} = A_{<i>}) \\ \wedge \ \mathrm{isConvex}(C_{<k>}) \ \wedge \ \bigwedge_{i=k+1}^{i=n} \mathrm{isSingleton}(C_{<i>})$$

The predicate isConvex(S) is true *iff* the set S is convex, and the predicate isSingleton(S) is true iff S contains a single element.

To permit runtime evaluation when necessary, we reduce each of the tests to a satisfiability question for which we can synthesize an equivalent conditional to be tested at run time. The first condition reduces to testing if the set $(\mathrm{spanOfDim}(A, i) - \mathrm{project}(C, i))$ is not satisfiable. The predicate isConvex(S) reduces to testing if the set $(\mathrm{convexHull}(S) - S)$ is not satisfiable. The predicate isSingleton(S) also reduces to a satisfiability test, but we omit the details here.

To avoid evaluating $O(n^2)$ predicates at compile-time, we use a single scan of the dimensions (leftmost first) to find the first dimension k for which $C_{<k>} \neq A_{<k>}$, and check the predicates for $k \ldots n$. If these predicates cannot be proven, we synthesize code to repeat this scan and check at runtime, when it can be done precisely by evaluating at most $n + 2$ predicates. In general, this test can be performed much faster than packing a buffer of modest size. If the decision must be made at runtime, we generate two versions of code (with and without buffer packing). This approach, based on explicit integer sets, enables us to exploit in-place communication for arbitrary communication sets, independent of data layouts and communication patterns.

2.5 Implementing Loop-Splitting For Reducing Communication Overhead

Previous researchers [22, 30] have suggested iteration reordering techniques to ameliorate two types of communication overhead: the cost of referencing buffered non-local data, and the latency of communication. Both techniques involve splitting a loop to separate the iterations that access only local data from those that may access non-local data. Buffer access costs arise because a run-time check may be needed to access local or non-local data when appropriate. After splitting the local and non-local iterations, no checks are needed for references in the local iterations. The latency of communication can be (partly) hidden by splitting because communication required for non-local iterations can be overlapped with local computation.

The only implementation of this approach we know of is in Kali [22], where the authors used set equations to explain the optimization but used pattern-based analysis to derive the iteration sets for a few special cases. This approach is only practical for a small number of special cases, and may be even more limited when statements in a loop have different CPs.

We extend the equations in [22] to apply to arbitrary sets of references, and any CP in our CP model, using the sets and mappings described in previous sections. We only describe loop-splitting for communication

Inputs:

$CPMap_s$	:	CP map for each statement s in a loop nest
$ReadRefs, WriteRefs$	:	non-local read and write references in the loop nest
Ref_r	:	map representing reference r, $\forall r \in CommRefs = ReadRefs \cup WriteRefs$
$CPMap_r$	:	CP map for reference r
$nlDataSet_r(m)$	:	set of non-local data accessed by processor m (myid)

Algorithm:

$$
\begin{aligned}
cpIterSet_r &= CPMap_r(\{m\}) \\
nlIters_r &= {Ref_r}^{-1}(nlDataSet_r(m)) \bigcap cpIterSet_r \\
nlReadIters &= \textstyle\bigcup_{r \in ReadRefs} NLIters_r \\
nlWriteIters &= \textstyle\bigcup_{r \in WriteRefs} NLIters_r \\
nonLocalIters &= nlReadIters \bigcup nlWriteIters \\
localIters &= \bigcup_{s\text{:loop stmts}} CPMap_s(\{m\}) - nonLocalIters \\
nlRWIters &= nlReadIters \bigcap nlWriteIters \\
nlROIters &= nonLocalIters - nlWriteIters \\
nlWOIters &= nonLocalIters - nlReadIters
\end{aligned}
$$

(a) Computing local/nonlocal iteration sets

```
SEND data for non-local reads
execute NLWOIters
execute LocalIters
RECV data for non-local reads
execute NLWOIters ⋃ NLRWIters
SEND data for non-local writes
RECV data for non-local writes
```

(b) Scheduling loop iterations

Figure 3: Loop splitting to overlap communication and computation.

overlap here, because it subsumes splitting for buffer access. Since, in dHPF, some iterations may also compute values for non-local data, we separate the set of iterations 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 as shown in Figure 3(a) for a loop-nest containing one or more potentially non-local references, $r_i, 1 \leq i \leq n$. The key step is computing the iterations that access non-local data for each given reference r. This can be directly computed from $nlDataSet_r(m)$ (obtained from the equations Figure 2), simply by using the mapping Ref_r^{-1} to derive the iterations that access these elements. $LocalIters$ includes all iterations assigned to processor m that are not in $nlIters_r$ for any r, and the other sets are computed simply by grouping $nlIters_r$ by read and write references.

We schedule the communication and computation for this loop nest in the sequence shown in Figure 3(b). Since the iterations of $NLRW$ (if any) must all be placed either before or after $LocalIters$, we simply place them after (merging them with $NLWOIters$). They could also be merged with $NLROIters$; a simple heuristic could be used to choose between the two alternatives. This sequence overlaps the read communication latency with the execution of $NLWOIters$ and $LocalIters$. It also overlaps write communication latency from $NLWOIters$ with the execution of $LocalIters$ and $NLROIters$.

Note that this form of splitting also subsumes splitting for non-local buffer access, since the local iterations are separated here as well. References in the local iterations do not need buffer-access checks, and a reference r in a non-local loop section (e.g., $NLROIters$) also does not need such checks if $nlROIters \subset nlIters_r$.

Code generation from these sets subsumes the operation of partitioning the loop by reducing the loop bounds, since each of the four loop sections is a subset of $\bigcup_{s:\text{loopstmts}} CPMap_s(\{m\})$. Therefore, code generation for loop-splitting is performed as part of the overall code generation framework for computation partitioning, described in Section 4.

3 Implementation of the Integer Set Framework

The sets and mappings that arise in our compiler require a fairly comprehensive integer set representation, including the ability to represent set union (or disjunctions of constraints), and set difference. For example, the union operation is required for a *CPMap* consisting of multiple individual ON_HOME terms, for the *CPMap* of a loop iteration containing multiple statements with different CPs, and for the data set of a communication event when communication for multiple references is coalesced.

Limited set representations such as Regular Section Descriptors [14] and Data Access Descriptors [4], as well as previous implementations of systems of linear inequalites supported by Fourier-Motzkin Elimination [2, 5, 1], do not support general non-convex sets set (and consequently, do not support general set union, difference, or complement operations). Their major advantage, however, is that they can support affine expressions with symbolic (i.e., unknown constant) coefficients.

In contrast, the Omega library from the University of Maryland (originally developed to support the Omega test for dependence analysis) provides a powerful implementation of a general class of integer set operations, including set union, using a generalization of Fourier elimination techniques [24, 19]. Omega also supports generation of efficient code for multiple iteration spaces (such as loops containing multiple statements with different CPs) [18]. While the underlying algorithms have poor worst-case performance, such cases appear very unlikely to occur in practice [24].[5]

From our perspective, a more significant limitation of the Omega library is that it does not permit unknown coefficients in affine constraints. These are required for all the HPF distributions when the number of processors is not known at compile time, for the cyclic(k) distribution with unknown k, and for loops with unknown strides. We have used the Omega library for our implementation, and extended our framework to circumvent the limitation on symbolic parameters in data layouts, as described below. Loops with unknown strides are not supported by our framework, 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, [20]), or an inspector-executor approach.

In the presence of a symbolic number of processors or symbolic k, we use a virtual processor (VP) model that naturally matches the semantics of templates in HPF [15]. We also add an additional optimization step to reduce runtime overhead in the resulting code. This optimization was also used by Gupta et al. [12] but was based on detailed analysis of specific patterns, whereas we use a general integer-set-based algorithm, described below.

For our VP model, we use 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. We construct the *Layout* mapping as a map from VPs to data elements. All the analyses described in the previous sections operate unchanged on the virtual processor domain. 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 representative physical processor `myid`. Also, when generating code for communication, we must aggregate the messages to all the VPs belonging to the same physical processor.[6]

[5] We are investigating approaches for detecting and avoiding such cases within the compiler.

[6] We are currently implementing these extra code generation steps. We have tested the virtual processor model using the Omega calculator (an interactive front end to the Omega library), and incorporated the model into the integer set framework

Inputs: Same as in Figure 2
Algorithm:

$$
\begin{aligned}
busyVPSet_r &= Domain(CPMap_r)\\
NLDataAccessed_r &= \begin{cases} DataAccessed_r - Layout_{Ar} & \text{if } r \in ReadRefs\\ DataAccessed_r \cap Layout_{Ar} & \text{if } r \in WriteRefs \end{cases}\\
allNLDataSet_r &= NLDataAccessed_r(busyVPSet_r)\\
vpsThatOwnNLData_r &= Layout^{-1}_{Ar}(allNLDataSet_r)\\
vpsThatAccessNLData_r &= Domain(NLDataAccessed_r)\\
activeSendVPSet &= \bigcup_{r \in ReadRefs} vpsThatOwnNLData_r \bigcup \bigcup_{r \in WriteRefs} vpsThatAccessNLData_r\\
activeRecvVPSet &= \bigcup_{r \in ReadRefs} vpsThatAccessNLData_r \bigcup \bigcup_{r \in WriteRefs} vpsThatOwnNLData_r
\end{aligned}
$$

(a) Equations for computing the active virtual processors

```
real A(1:100)
processors PA(P1,P2)
template T(100,100)
align A(i,j) with T(i,j)
distribute t(cyclic,cyclic) onto PA
...
do i = PIV+1, 100
  do j = PIV+1, 100
    ! ON_HOME { A(i,j) }
    A(i,j) = ··· + A(PIV,j)
  enddo
enddo
```

(b) HPF code for Gauss loop

$$
\begin{aligned}
vpArray &= \{[v_1, v_2] : 1 \le v_1, v_2 \le 100\}\\
loop &= \{[i, j] : PIV + 1 \le i, j \le 100\}\\
CPMap &= \{[v_1, v_2] \to [i, j] : i = v_1 \wedge j = v_2 \wedge PIV < v_1, v_2 \le 100\}\\
ReadRefs &= \{[r] : r = \text{'}\mathtt{A(PIV,j)}\text{'}\}\\
busyVPSet &= \{[v_1, v_2] : PIV < v_1, v_2 \le 100\}\\
NLDataAccessed &= \{[v_1, v_2] \to [PIV, v_2] : PIV < v_1, v_2 \le 100\}\\
activeSendVPset &= \{[v_1, v_2] : v_1 = PIV \wedge PIV < v_1, v_2 \le 100\}\\
activeRecvVPset &= busyVPs
\end{aligned}
$$

(c) Active virtual processors in Gauss loop

Figure 4: Active virtual processors for computing, sending and receiving

The additional optimization mentioned above is used to reduce the runtime overhead of the extra virtual processor loop. With this optimization, we compute the entire set of VPs that are active in each given operation (a partitioned computation, sending data, or receiving data). In a partitioned loop, for example, some VPs for a given physical processor may have no iterations to execute. Given these active VP sets, each physical processor only needs to iterate over the subset of its VPs that are active in the operation, and not all its VPs, eliminating the need for runtime checks inside the virtual processor loop.

The set of active VPs for each problem is computed as shown in Figure 4(a). The active VPs (denoted $busyVPSet$) for any partitioned computation is simply the domain of the $CPMap$. The active VPs that must send or receive data (for each logical communication event) can be computed using a map from processors to non-local data referenced by each processor ($NLDataAccessed$). As explained in Section 2.3, however, this map cannot be computed when multiple processor arrays appear in the domain of the same CP. In such a case, which we expect to be rare, we would fall back on using runtime checks.

The results of these equations are illustrated for the Gaussian Elimination example in Figure 4(b,c), where the reference to the pivot row, $A(PIV, j)$, requires off-processor data. The $busyVPSet$ reflects that only VPs corresponding to the lower, right portion of the matrix A are active. $activeSendVPSet$ and $activeRecvVPSet$ indicate that only VPs owning elements in the pivot row (PIV) must send any data, but

implementation, including the equations to optimize runtime overhead.

all VPs active in the computation (busyVPs) must receive non-local data. In practice, we can generate code so that only one VP per physical processor will receive each such element.

In addition to using the virtual processor model, one additional implementation detail arises in the presence of multiple processor arrays *and* a symbolic number of processors. The union operation used to compute $CPMap$ (Equation 1) cannot be performed directly when the arrays A_k are distributed on different processor arrays (even though these represent the same set of processors). (Effectively, each actual processor is represented by different tuple elements in the different domains). One alternative is to map processor indices in every processor array to a common 1-D processor array by linearizing the indices. However, when the number of processors in any dimension is symbolic, this mapping would require a product of two symbolics, which cannot be directly represented in Omega.

Instead, we perform the above union implicitly, simply by maintaining a list of the individual mappings, CP_k, using a data structure called a TupleList. The TupleList implementation supports all the standard operations on sets and mappings, including inverse, intersection, union, difference, domain, range, and composition with a set or mapping. Further operations involving $CPMap$ operate directly on TupleLists instead of single sets or mappings. (The number of entries in such a list is always bounded above by the highest number of items n in the relevant CPs.) In most cases, we eventually are interested in the range of the mappings in the resulting TupleList (e.g., for the iteration spaces in a computation partitioning or the data elements in a message sent to each processor). Note that only the domains of the entries in the $CPMap$ TupleList are of different ranks; the ranges all have identical ranks. The range of such a TupleList can therefore be directly represented as a single set. Therefore, no precision is eventually lost, even through entire sequences of operations on TupleLists. The primary limitation arises in computing the *domain* of such a TupleList (e.g., to compute the set of processors that must execute a SEND). This set cannot be computed explicitly *if* there actually exist multiple processor arrays used within the same CP, and the number of processors in any dimension is symbolic. We expect the former condition to be infrequent in practice, and handle it with some runtime support, as described in Section 2.3.

The techniques described above make it possible for us to take advantage of the capabilities of the integer-set framework without being unduly limited by the limitations on the representation of symbolic quantities. The additional complexity introduced by these techniques are largely encapsulated in the implementation of the framework, and (we believe) are greatly outweighed by the analysis capabilities and flexibility the framework provides.

4 Code generation for general computation partitions

With the general CP model of Section 2.2, efficient partitioned code may require a loop to be split into multiple convex sections in order to avoid expensive runtime checks. This introduces a tradeoff between bottom-up and top-down code generation: a top-down strategy provides precise information about enclosing scopes (because they will have already been partitioned), but may require many more applications of the partitioning algorithm than a bottom-up strategy. For example, in a 2-deep loop-nest where each loop is eventually split into two sections, a top-down strategy would invoke the loop partitioning algorithm three times. A bottom-up strategy would invoke it only twice, but at the inner level, it would have no information about the two separate sections that will result at the outer level. We use a two-pass algorithm to resolve this tradeoff, using the more efficient bottom-up strategy to synthesize a correct (and fairly efficient) partitioned SPMD node program, and then use a much simpler "top-down" algorithm to further simplify the resulting code.

4.1 Realizing computation partitions

To partition a procedure based on CP assignments, we perform a post-order traversal of a tree-based data abstraction of the procedure, applying code generation transformations as needed at each node in the tree. A node in the tree represents a single DO statement, a single branch of a conditional branch, or a sequence of simple statements with a uniform CP (each of the former two is referred to as a "scope"). Different code generation algorithms can be applied at each node in the tree. For example, for a loop, we currently support two strategies: simple bounds reduction, or loop-splitting (as in Section 2.5) combined with bounds reduction for the individual loop sections. Two other alternatives applicable to loops with irregular data layouts or references, namely runtime resolution and an inspector-executor strategy, are under development.

To partition a single loop or loop-nest with regular computation partitionings that can be represented explicitly as integer mappings, we Omega library's algorithm for code generation with multiple iteration spaces [18]. The function reduces loop bounds and also lifts guards out of inner loops when multiple statements with non-overlapping iteration spaces exist. An important optimization we apply is to provide available information about enclosing scopes so as to simplify the resulting code. For example, consider the loop nest in Figure 5. To generating code for the inner (j) loop alone, we fix the value of i at a symbolic value I. We then use $\mathit{LoopCP}_2(I)$ as "known" information because the constraints in $\mathit{LoopCP}_2(i)$ will be enforced when partitioning the enclosing i loop. Therefore, the inner loop is partitioned using the operation $\mathrm{CodeGen}(CP_2(I,j), CP_3(I,j) \mid \mathit{LoopCP}_2(I))$ (see Appendix A). Then, the outer loop is partitioned using $\mathrm{CodeGen}(CP_1(j), \mathit{LoopCP}_2(i) \mid \mathit{LoopCP}_1())$.

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

$$
\begin{aligned}
\mathit{LoopCP}_1() &= \mathrm{Project}(\, CP_1(i) \cup \mathit{LoopCP}_2(i),\ \{\,[i] : 1 \le i \le N\,\}\,) \\
CP_1(i) &= \{\,[i,j] : \text{myid owns } A_1(f_1(i))\} \\
\mathit{LoopCP}_2(i) &= \mathrm{Project}(\, CP_2(i,j) \cup CP_3(i,j),\ \{\,[j] : 1 \le j \le M\,\}\,) \\
CP_2(i,j) &= \{\,[i,j] : \text{myid owns } A_2(f_2(i,j))\,\} \\
CP_3(i,j) &= \{\,[i,j] : \text{myid owns } A_3(f_3(i,j))\,\}
\end{aligned}
$$

Figure 5: Example showing iteration sets constructed for code generation.

4.2 Control flow simplification

The bottom-up algorithm for realizing CPs can result in excess code in inner scopes, because of imperfect information regarding the partitioned code of outer scopes. Furthermore, when communication statements are placed inside a loop due to a loop-carried dependence, the communication code generated from $\mathit{SendCommMap}$ and $\mathit{RecvCommMap}$ (Section 2.3) includes control conditions to ensure that processors execute only their assigned communication. These conditions may also include superfluous guards, given the constraints of enclosing partitioned loops.

To illustrate the problem, Figure 6(a) shows source code for a loop from the Erlebacher benchmark and Figure 6(b) shows the skeletal SPMD code resulting from bottom-up code generation followed by communication code generation. In the generated code, many of the guards are infeasible or tautological because the outer j loop is split into three sections ($j = 16pmyid_1 + 16$, $\{\, 16pmyid_1 + 15 \ge j \ge 16pmyid_1 + 1\}$, and $j = 16pmyid_1$) to avoid runtime checks on the communication calls, but this information was not available when partitioning the i loop.

We can exploit constraint information from outer scopes to simplify control flow in inner ones. A reverse-postorder traversal of the control-dependence graph (ancestors before descendants) computes constraints at each node that are guaranteed by constraints (conditionals and loop ranges) of its control dependence ancestors. These constraints are represented directly by integer sets of rank 0, where non-affine expressions

```
      parameter (N=64)
      real a(N), b(N), f(N,N)
CHPF$ processors p(4)
CHPF$ template t(N,N)
CHPF$ distribute t(*,block) onto p
CHPF$ align f(i,j)     with t(i,j)
CHPF$ align a(j)       with t(*,j)
CHPF$ align b(j)       with t(*,j)
      do j=N,2,-1
        do i=1,N
          f(i,j)=(f(i,j)-a(j)*f(i,j+1))*b(j)
        enddo
      enddo
```

(a) HPF source for a fragment from the Erlebacher benchmark.

```
j = 16 * pmyid1 + 16
if (pmyid1 .le. 2) then
  RECV
do i = 1, 64
  ...
do jminus = -(16 * pmyid1) - 15,
    min(-(16 * pmyid1) - 1, -2)
  j = -jminus
  do i = 1, 64
    ...
if (pmyid1 .ge. 1) then
  j = 16 * pmyid1
  SEND
```

(c) Generated code after simplification.

```
j = 16 * pmyid1 + 16
if (16 * pmyid1 .ge. j - 15) then
  do i = 1, 64
    ...
if (16 * pmyid1 .eq. j - 16 .and. pmyid1 .le. 2) then
  RECV
if (16 * pmyid1 .le. j - 16) then
  do i = 1, 64
  ...
do jminus = -(16 * pmyid1) - 15,
    min(-(16 * pmyid1) - 1, -2)
  j = -jminus
  if (16 * pmyid1 .eq. j .and. pmyid1 .ge. 1) then
    SEND
  if (16 * pmyid1 .ge. j - 15) then
    do i = 1, 64
      ...
  if (16 * pmyid1 .eq. j - 16 .and. pmyid1 .le. 2) then
    RECV
  if (16 * pmyid1 .le. j - 16) then
    do i = 1, 64
      ...
if (pmyid1 .ge. 1) then
  j = 16 * pmyid1
  if (16 * pmyid1 .eq. j .and. pmyid1 .ge. 1) then
    SEND
```

(b) Generated code before simplification.

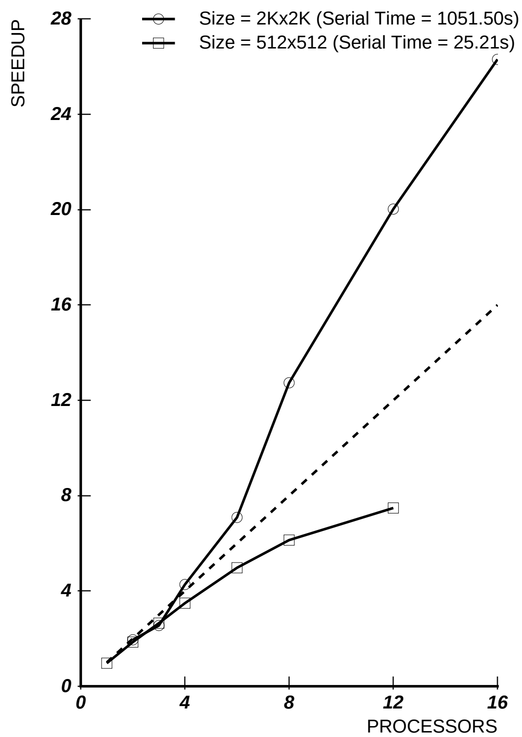
Figure 6: Control-flow simplification for a pipelined loop from Erlebacher.

are simply reduced to conditions on synthetic global variables. Then, in a postorder traversal of the graph, we use Codegen($S \mid C$) to simplify or eliminate code at each control-flow node, where S denotes the original constraints for that node and C the guaranteed constraints from CD ancestors. The final, simplified code for the Erlebacher example loop is shown in Figure 6(c).

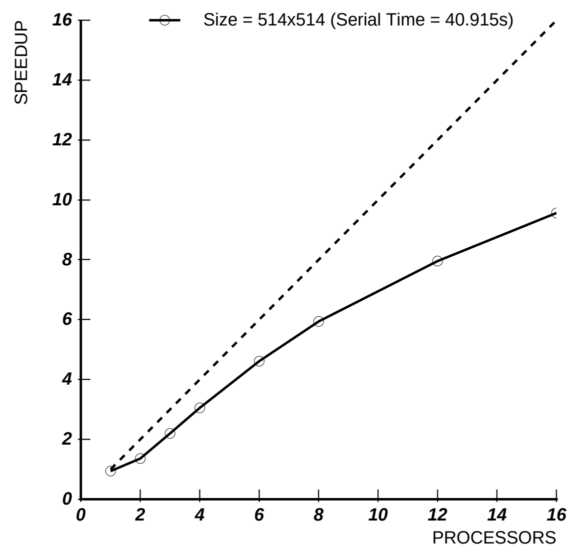
It is interesting to note in the Erlebacher example that the final placement of the communication code is exactly what the optimization known as vector message-pipelining aims to achieve [30]. This optimization moves pipelined shift communication occuring in boundary iterations of a partitioned loop out of the loop in order to eliminate runtime checks on the communication statements. This optimization is quite complex to implement in a general way using pattern-based techniques, but in dHPF it is easily obtained as a direct result of bounds-reduction followed by control-flow simplification (although neither step individually would achieve this effect, as is evident from Figure 6).

5 Preliminary Results

With two exceptions, all the analyses and optimizations described in the previous sections have been fully implemented. The exceptions are code-generation support (to generate two-version code) for finding in-place communication at runtime, and code-generation support for the virtual processor model. The dHPF compiler currently generates Fortran 77 or Fortran 90 SPMD programs using MPI [27] for communication.



(a) Speedups for Jacobi



(b) Speedups for Tomcatv

Figure 7: Preliminary speedups for two benchmarks on SGI Power Challenge Cluster / MPI

The goal of the work presented so far is primarily to demonstrate that it is practical and extremely powerful to use an integer set framework for implementing key analyses and optimizations in an HPF compiler. In particular, this approach greatly simplifies the implementation (even with a very general computation partitioning model) and simultaneously enhances the generality and flexibility of these analyses and optimizations. Many of the specific optimizations described here are not new (although few if any compilers have implemented more than a few of these in a general way), and therefore evaluating the impact of these optimizations on applications is not directly relevant here (but is a subject of our current work).

We present some experimental results to show that the overall implementation of the optimizations is practical and effective enough to achieve moderate-to-good speedups for realistic kernels and benchmarks. These results are preliminary and no tuning has been performed so far on the compiler to improve the performance of the generated parallel code. We present results for two benchmarks: Tomcatv—a mesh generation code from the SPEC92 benchmark suite, and Jacobi—a simple 4-point stencil kernel with a convergence loop. We distributed the key arrays `(block,*)` in the former and roughly square `(block,block)` in the latter. The number of physical processors was specified to dHPF at compile time. The compiler required about 3 minutes on a SparcStation-10 to compile Tomcatv with all optimizations, and much less for Jacobi. The Fortran code generated by dHPF was compiled with the SGI IRIX 6.2 `f77` compiler at optimization level -O3. Figure 7 shows the resulting speedups on a cluster of four SGI Power Challenge-L multiprocessors with four-processors each, interconnected by a HIPPI switch.

For Tomcatv, the compiler provides moderate speedup up to 16 processors, partly because of extensive communication as well as two global reductions within the main computational loop. The compiler selected non-owner computes computation partitions for two array assignment statements, and generated different partitionings for different statements in several of the loops. Nevertheless, the scalar efficiency of the generated code is quite good (partly shown by the speedup of 0.94 on 1 processor). There were no opportunities for in-place communication with the (block, *) distribution, but nearly all data received was unpacked into overlap areas [9] which reduces guard overhead for using the data. The compiler also recognized and implemented two reductions. We considered these results to be acceptable before any compiler tuning, for a cluster system with relatively high communication overhead (e.g., a four-byte node-to-node message costs about 119 μs).

For Jacobi, the speedup only scales linearly up to $P = 8$ with a small problem size, but shows roughly the expected linear speedup for a larger problem. The latter curve shows superlinear speedup beginning at 6 processors as the problem begins to fit entirely in cache. The optimizations applied here included loop splitting for communication/computation overlap, bounds reduction, using in-place communication (with overlap areas), and recognizing the reduction for the convergence loop. Loop splitting does not realize much overlap because our compiler currently uses an MPI primitive with sender-side buffering that prevents overlap of communication and computation at run time.

6 Related Work

As explained in the Introduction, most research and commercial data-parallel compilers to date use pattern-based approaches for implementing basic communication and iteration set analysis [23, 6, 22, 28, 30, 31, 10, 11, 13, 7, 12]. This is a fundamentally different approach from that taken in this paper, and its strengths and weaknesses have been discussed in the Introduction. 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, 8, 20]). These techniques likely provide more efficient support for cyclic(k) distributions 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.

The previous work on using linear inequalities and Fourier-Motzkin Elimination [26] for code generation share our goal of improving the generality, the level of abstraction, and the quality of code in data-parallel compilers. Three groups (Ancourt et al. [2], the Paradigm compiler group [29] and SUIF [1]) applied these techniques to support code generation for communication and iteration sets described by linear inequalities. The representations used by these groups were unable to describe general non-convex sets, which limited the scope of their techniques (e.g., it precluded addressing problems such as coalescing communication for arbitrary affine references or splitting iterations sets). The SUIF work [1] assumes that all statements in a loop have identical computation partitionings. In contrast, we support both communication analysis and code generation for a much more general computation partitioning model than previous compilers. Our virtual processor model for distributions on symbolic numbers of processors is similar to the SUIF model, although they use it for any cyclic(k) distribution. The latter work also describes how several basic communication analyses and optimizations can be performed using their representation based on linear inequalities, such as message aggregation and redundant communication elimination. Unlike these previous projects, however, we develop a descriptive equational framework to support general communication analyses and optimizations, and show how to express several important optimization steps in this framework.

Finally, Pugh et al. developed efficient techniques for Presburger simplification and for code generation with multiple mappings, and incorporated these in the Omega library. In this paper, we have described how this technology can be applied to significantly enhance the computation partitioning, communication analysis, and code generation capabilities of a data-parallel compiler.

7 Conclusions

In this paper, we have described an integer-set-based approach for analysis and code generation for data parallel programs. Our implementation of this approach in the Rice dHPF compiler has shown that the approach greatly simplifies the implementation of many key analyses and optimizations (even with a general computation partitioning model), and yet enhances their generality and flexibility. To implement this framework in dHPF, we used an integer-set package (the Omega library) which provides powerful simplication and code generation operations for integer-sets, but extended the framework to circumvent limitations imposed by the package in representing symbolic data distribution parameters.

We used our framework to construct a number of key analyses and optimizations including computing communication, iteration, and processor sets for our general computation partitioning model, performing message vectorization and coalescing for general patterns and array references, more accurately detecting when data can be communicated "in-place", and splitting loops for reducing buffer access overheads and for overlapping communication with local computation. Finally, we have described an efficient hierarchical code generation strategy that makes it possible to harness complex algorithms for synthesizing efficient partitioned code for our general computation partitioning model.

Perhaps most important, we believe our equational framework and supporting program analysis technology can greatly simplify the development of sophisticated new optimizations expressible as integer sets operations. Thus, the framework provides a simple, uniform, and powerful foundation for analysis, optimization and code generation of data-parallel programs.

Acknowledgements

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 the integer set equations described here. The dHPF compiler is the result of the collective efforts of all the members of the dHPF compiler group. Lisa Thomas implemented the algorithm to detect in-place communication from integer sets. Sarita Adve, Robert Fowler and Ken Kennedy provided valuable comments on earlier drafts of this paper. This work has been supported in part by DARPA Contracts DABT63-92-C-0038 and F30602-96-1-0159 and a State of Texas Advanced Technology Program Award 003604-017.

References

[1] 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.

[2] 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.

[3] 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.

[4] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[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] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[7] 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.

[8] 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.

[9] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[10] 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.

[11] 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.

[12] 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, 1 February 1996.

[13] 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.

[14] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[15] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[16] 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.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[18] 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.

[19] 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.

[20] 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.

[21] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[22] 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.

[23] 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.

[24] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[25] 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.

[26] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, Chichester, Great Britain, 1986.

[27] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, , and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.

[28] 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.

[29] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[30] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[31] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

Appendix A. Definitions of integer set operations

Some of the less common operations on sets and maps are explained below. We follow the notation in the Omega package [19] to define these operations, using the following sets and maps:

$$\begin{aligned}
S1 &= \{[i_1 \ldots i_n] : f_{S1}(i_1 \ldots i_n)\} \\
S2 &= \{[i_1 \ldots i_k] : f_{S2}(i_1 \ldots i_k)\} \\
R1 &= \{[i_1 \ldots i_n] \rightarrow [j_1 \ldots j_m] : f_{R1}(i_1 \ldots i_n, j_1 \ldots j_m)\} \\
R2 &= \{[i_1 \ldots i_m] \rightarrow [j_1 \ldots j_p] : f_{R2}(i_1 \ldots i_m, j_1 \ldots j_p)\}
\end{aligned}$$

$$\begin{aligned}
\text{Composition} : R1 \circ R2 &\equiv \{[i_1 \ldots i_n] \rightarrow [j_1 \ldots j_p] : \exists \alpha_1 \ldots \alpha_m \ s.t. \ f_{R1}(i_1 \ldots i_n, \alpha_1 \ldots \alpha_m) \wedge f_{R2}(\alpha_1 \ldots \alpha_m, j_1 \ldots j_p)\} \\
\text{Composition} : R1(S1) &\equiv \{[j_1 \ldots j_m] : \exists \alpha_1 \ldots \alpha_n \ s.t. \ f_{R1}(\alpha_1 \ldots \alpha_n, j_1 \ldots j_m) \wedge f_{S1}(\alpha_1 \ldots \alpha_n)\} \\
\text{Project}(S1, S2) &\equiv \{[i_1 \ldots i_{n-k}] : \exists \alpha_{n-k+1} \ldots \alpha_n \ s.t. \\
&\qquad f_{S1}(i_1 \ldots i_{n-k}, \alpha_{n-k+1} \ldots \alpha_n) \wedge f_{S2}(\alpha_{n-k+1} \ldots \alpha_n)\}, k \leq n \\
\text{RestrictVars}(S1, v) : &\equiv \{[i_1 \ldots i_n] : i_1 = I_1 \wedge \ldots \wedge i_v = I_v \\
&\qquad \wedge f_{S1}(I_1 \ldots I_v, i_{v+1} \ldots i_n)\}, \quad 1 \leq v \leq n \\
&\quad \text{(where } I_1 \ldots I_v \text{ are dummy symbolic variables)} \\
\text{RestrictRangeVars}(R1, v) : &\equiv \{[i_1 \ldots i_n] \rightarrow [j_1 \ldots j_m] : j_1 = J_1 \wedge \ldots \wedge j_v = J_v \\
&\qquad \wedge f_{R1}(i_1 \ldots i_n, J_1 \ldots J_v, j_{v+1} \ldots j_n)\}, \quad 1 \leq v \leq n \\
&\quad \text{(where } J_1 \ldots J_v \text{ are dummy symbolic variables)}
\end{aligned}$$

Codegen($S1 \ldots Sv \mid Known$) : $S1 \ldots Sv$: iteration spaces for v statements.
$Known$ (a set of rank 0) : Constraints on global variables in $S1 \ldots Sv$ that will be externally enforced.
Synthesizes a code sequence to enumerate the tuples in $S1 \ldots Sv$ in lexicographic order, where the same tuple in different sets is ordered as: $(\underline{i} \in Sj) \prec (\underline{i} \in Sk), j < k$.

Codegen($S1 \mid Known$), $n = 0$: Synthesizes an IF statement that executes its body if the condition f_{S1} is satisfied (assuming the constraints in Known are externally enforced)