Using Integer Sets for Data-Parallel Program Analysis and Optimization*

Vikram Adve John Mellor-Crummey
Department of Computer Science
Rice University
{adve,johnmc}@cs.rice.edu

Abstract

In this paper, we describe our experience with using an abstract integer-set framework to develop the Rice dHPF compiler, a compiler for High Performance Fortran. We present simple, yet general formulations of the major computation partitioning and communication analysis tasks as well as a number of important optimizations in terms of abstract operations on sets of integer tuples. This approach has made it possible to implement a comprehensive collection of advanced optimizations in dHPF, and to do so in the context of a more general computation partitioning model than previous compilers. One potential limitation of the approach is that the underlying class of integer set problems is fundamentally unable to represent HPF data distributions on a symbolic number of processors. We describe how we extend the approach to compile codes for a symbolic number of processors, without requiring any changes to the set formulations for the above optimizations. We show experimentally that the set representation is not a dominant factor in compile times on both small and large codes. Finally, we present preliminary performance measurements to show that the generated code achieves good speedups for a few benchmarks. Overall, we believe we are the first to demonstrate by implementation experience that it is practical to build a compiler for HPF using a general and powerful integer-set framework.

1 Introduction

Data-parallel languages such as High-Performance Fortran (HPF)[20] aim to provide a simple, portable, abstract programming model applicable to a wide variety of parallel systems. To achieve wide acceptance, a data-parallel language requires parallelizing compilers that can provide consistently high performance for a broad class of applications. Current commercial compilers for HPF [8, 12, 14] typically yield widely varying performance for different applications, even among "regular" data-parallel applications on message passing systems. (Regular applications are those with statically analyzable data access patterns and, usually, well-balanced computational costs.)

Most research and commercial data-parallel compilers to date [22, 7, 21, 29, 16, 11, 12, 14, 8, 13, 32, 33] (including the Rice Fortran 77D compiler) perform communication analysis and code generation by considering specific combinations of the form of references, data layouts and computation partitionings. Such case-based analysis has been the principal implementation technique for data-parallel compilation systems because it provides a practical and conceptually simple strategy for developing compilers, and it can be relied on to yield excellent performance for common cases. For example, even though Hiranandani et al. [16] and Koelbel [21] both formulated computation partitionings and communication analysis in terms of abtract integer-set operations (assuming the owner-computes rule [26]), they used case-based approaches in their implementations of the Fortran D and Kali compilers. The case-based approach, however, provides poor performance for cases that have not been explicitly considered. Perhaps more significantly, such 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 to offer consistently high performance.

A few researchers have used analysis techniques based on linear inequalities, which enable a more general and flexible approach than case-based analysis for implementing data parallel languages [3, 6, 2]. This work has focused on computing local communication and iteration sets, and performing code generation from these sets using Fourier-Motzkin elimination (FME) [4]. Amarasinghe and Lam [2] also describe how inequalities and FME can support array dataflow analysis and a few specific communication optimizations. A limitation of the linear inequality representations used by these groups is that they cannot represent general non-convex sets. These representations therefore preclude optimizations such as coalescing communication for arbitrary affine references [2], non-local index-set splitting [21], or the use of a general computation partitioning model such as that used in our work.

Recently, Pugh et al. have reported important progress in developing efficient algorithms for manipulating and generating code from general non-convex sets [25, 18]. This provides a potentially important new tool for developing parallelizing compilers, but leaves open two key questions: Is it practical to use these techniques for real programs? And what challenges must be addressed in formulating and implementing the key analysis and optimizations problems using such an approach?

In this paper, we describe our experience with using a

*This work has been supported in part by DARPA Contract DABT63-92-C-0038, the Texas Advanced Technology Program Grant TATP 003604-017, an NSF Research Instrumentation Award CDA-9617383, and sponsored by DARPA 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 DARPA and Rome Laboratory or the U.S. Government.

$data_k$:	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 $\underline{p} \in proc_n\}$
RefMap:	$loop_k \rightarrow data_n$: $\{[\underline{i}] \rightarrow [\underline{a}]$: 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 $\underline{p} \in proc_n\}$

Figure 1: Primitive sets and mappings for compiling data-parallel programs.

general integer set representation to develop the Rice dHPF compiler, a compiler for High Performance Fortran. In particular, the compiler is based on an abstract equational framework that expresses data parallel program analyses and optimizations in terms of operations on symbolic integer sets. Using this framework, we have devised and implemented simple, concise, yet general, equational formulations of the major partitioning and communication analyses as well as a number of important optimizations. Because of the simplicity of these formulations, it has been possible to implement a comprehensive collection of advanced optimizations in dHPF. In most cases, our implementations of the optimizations are more general than in previous compilers. Furthermore, it has been possible to support all these analyses and optimizations in the context of a more general computation partitioning (CP) model than that used in previous compilers.

The specific contributions of this work are as follows. First, we describe the formulation and implementation of key data-parallel program analyses and several optimizations using integer set equations. These include:

- static loop partitioning and communication analysis for a general computation partitioning (CP) model;
- communication vectorization for arbitrary regular communication patterns;
- message coalescing for arbitrary affine references to an array;
- a powerful class of loop-splitting transformations that support several optimizations, including overlapping communication with computation *within* a single loop-nest, and reducing the overhead of accessing buffered non-local data; and
- a combined compile-time/run-time algorithm to reduce explicit data copies for a message.

All of these analyses and optimizations have been implemented in the dHPF compiler. Of these, the CP model, the loop-splitting based optimizations, the analysis for buffering non-local data, and message coalescing are all more general than in previous compilers.

Second, a fundamental limitation of a general integer-set based representation is that set constraints containing products of integer variables yield problems that are undecidable [17]. The primary source of such symbolic product terms is the HPF `distribute` directive with unknown processor counts or block sizes. We describe a natural extension to our framework based on a virtual processor model that supports these symbolic terms without requiring any changes to the set formulations for the above optimizations. A key component of this extension is an integer-set algorithm and associated code-generation strategy that eliminates or reduces runtime checks by restricting the computation and communication to the *active* virtual processors. This extension makes it possible to compile HPF programs for an unspecified number of processors, and we present experimental results to show that there is little or no difference in compile-time for a symbolic than for a constant number of processors, even on fairly large and complex codes such as the NAS SP application benchmark.

Finally, we show experimentally that the set representation is not a dominant factor in compile times on both small and large codes. We also present preliminary results to show that the set-based implementation of the optimizations is practical and achieves good speedups for realistic kernels and benchmarks.

The next section describes our integer-set based equational framework. Section 3 describes the formulation of analyses and optimizations. Section 4 describes the framework extension for a symbolic number of processors. Section 5 describes a few key implementation considerations. Section 6 examines the compile-time cost of using this approach. Section 7 presents preliminary results on the performance of benchmark codes. Section 8 reviews related work. Section 9 states our conclusions.

2 A 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, such as HPF, operates primarily on three types of tuple spaces, and the three pairwise mappings between these tuple spaces ([2, 15, 21]). These are shown in figure 1.[1] 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 as tuples of rank zero. Hereafter, the terms "array" and "iterations of a statement" imply scalars and outermost statements as well.

The sets loop and proc and the mappings Layout and RefMap are computable directly from the source program and form the primary inputs for further analyses. Figure 2 illustrates these primitive sets and mappings for an HPF code fragment. Layout is computed from Align, which represents the alignment of an array with a template, and Dist, which represents the distribution of a template on a physical processor array. The iteration set, loop, follows directly from the loop bounds. The ON_HOME CP notation and construction of CPMap are described in the following section.

To implement analysis and optimization in dHPF as operations on these symbolic sets and mappings, we require an integer set package that supports operations including intersection, union, difference, domain, range, composition, and projection. For this purpose, we use the Omega Library developed by Pugh et al at the University of Maryland [17]. Omega supports representation and manipulation of (potentially non-convex) integer sets described by Presburger formulae, using algorithms based on Fourier-Motzkin Elimination (FME) [25]. This approach has both advantages and

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

```
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
          ! ON_HOME B(j-1,i)
    A(i,j) = B(j-1,i)
  enddo
enddo
```

symbolic N

$$
\begin{aligned}
\text{proc} &= \{[p] : 1 \le p \le 4\} \\
\text{Align}_A &= \{[a_1, a_1] \rightarrow [t_1, t_2] : t_1 = a_1 + 1 \wedge t_2 = a_2\} \\
\text{Align}_B &= \{[b_1, b_2] \rightarrow [t_1, t_2] : t_2 = b_1\} \\
\text{Dist}_T &= \{[t_1, t_2] \rightarrow [p] : 25p + 1 \le t_2 \le 25(p+1) \wedge 0 \le p \le 3\} \\
\text{Layout}_A &= \text{Dist}_T^{-1} \circ \text{Align}_A^{-1} \\
&= \{[p] \rightarrow [a_1, a_2] : max(25p+1, 1) \le a_2 \le min(25p+25, 100) \wedge \\
&\qquad 0 \le a_1 \le 99\} \\
\text{Layout}_B &= \text{Dist}_T^{-1} \circ \text{Align}_B^{-1} \\
&= \{[p] \rightarrow [b_1, b_2] : max(25p+1, 1) \le b_1 \le min(25p+25, 100) \wedge \\
&\qquad 1 \le b_2 \le 100\} \\
\text{loop} &= \{[l_1, l_2] : 1 \le l_1 \le N \wedge 2 \le l_2 \le N+1\} \\
\text{CPRef} &= \{[l_1, l_2] \rightarrow [b_1, b_2] : b_2 = l_1 \wedge b_1 = l_2 - 1\} \\
\text{CPMap} &= \text{Layout}_B \circ \text{CPRef}^{-1} \bigcap\nolimits_{\text{range}} \text{loop} \\
&= \{[p] \rightarrow [l_1, l_2] : 1 \le l_1 \le min(N, 100) \wedge \\
&\qquad max(2, 25p+2) \le l_2 \le min(N+1, 101, 25p+26)\}
\end{aligned}
$$

Figure 2: 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. (See Appendix A for notation.)

disadvantages compared to simpler set representations such as extended Rectangular Sections [16] or Data Access Descriptors [5]. In Section 4, we discuss the tradeoffs that arise and describe how we accommodate one of the key limitations of FME.

3 Optimizations using the Integer Set Framework

In this section, we describe analysis and optimizations using the integer set framework described in the previous section.

3.1 Computation Partitioning Model

A computation partitioning (CP) for a statement specifies which processor(s) must execute each dynamic instance of the statement. The compiler must support a flexible class of computation partitionings to enable choice of a partitioning well-suited to an application's needs.

Research and commercial HPF compilers primarily use the *owner-computes* rule [26] to assign CPs to statements. This rule specifies that a computation is executed by the owner of the value being computed. This rule, as well as other variants used in some compilers (e.g., decHPF [14] and SUIF [2]) can be expressed in terms of the processor(s) that own a particular set of data elements. In particular, for a statement enclosed in a loop nest with index vector $\underline{i}$, and for some variable A, the CP ON_HOME$\{A(f(\underline{i}))\}$, specifies that the dynamic instance of the statement in iteration $\underline{i}$ will be executed by the processor(s) that own the array element(s) $A(f(\underline{i}))$. This set of processors is uniquely specified by subscript vector $f(\underline{i})$ and the layout of array A at that point in the execution of the program. The SUIF compiler [2] further restricts all statements in a loop to have the same computation partition. The dHPF compiler supports a more general CP model in which a CP for a statement can be specified as the owner of one *or more* arbitrary data references, and each statement in a program may have its own CP. A statement's CP is specified by a union of one or more ON_HOME terms: $CP : \cup_{j=1}^{j=n} \text{ON\_HOME}\{A_j(f_j(\underline{i}))\}$. This *implicit* representation of a computation partitioning in dHPF supports arbitrary index expressions or any set of values in each index position in $f_j(\underline{i})$.

We convert the implicit CP form into an explicit integer tuple mapping, CPMap. This is possible when each subscript expression in $f_j(\underline{i})$ is an affine expression of the index variables, $\underline{i}$, with known constant coefficients, or is a strided range specifiable by a triplet *lb:ub:step* with known constant *step*. The overall mapping is a union of mappings for the individual ON_HOME terms:

$$\text{CPMap} = \bigcup_{j=1}^{j=n} (\text{Layout}_{A_j} \circ \text{Ref}_j^{-1}) \bigcap\nolimits_{\text{range}} \text{loop}$$

(The operator $\circ$ is defined in Appendix A.) Here, the mapping for a single term ON_HOME$\{A_j(f_j(\underline{i}))\}$ is a composition of the layout and reference mappings, restricted in range to the loop index space. CPMap explicitly specifies the processor assignment for the instance of a statement in loop iteration $\underline{i}$.

There are two key challenges in using such a general CP model: (a) generating efficient code to implement a chosen static loop partitioning, and (b) deriving symbolic data and processor sets for communication. We discuss code generation to support loop partitioning briefly below; computing communication sets is discussed in the next section.

The compiler generates a partitioned single program multiple data (SPMD) node program as specified by the CP for each statement. Briefly, the compiler uses a hierarchical code generation strategy that considers each scope (i.e., loop, conditional, or procedure) in isolation [1]. For efficiency, code generation proceeds in a depth first traversal over the tree of scopes in a procedure. For each scope, we compute a vector of sets, CPMap($\underline{m}$), one for each *statement group* in the scope. (A statement group is a sequence of consecutive statements with identical CPs. $\{\underline{m}\}$ is a singleton set representing the processor index vector for the fixed processor *myid*.) We then use Kelly, Pugh, and Rosser's algorithm for multiple-mappings code generation [18] (defined in Appendix B) to compute loop bounds and guards that enumerate the SPMD iteration space for each statement group in the scope. To avoid adding the same guards at multiple levels, we provide the iteration set of the immediately enclosing scope statement as the "known" parameter to Codegen, because those constraints will be enforced when generating code for the outer scope. In Section 5, we describe the techniques we use in the implementation to minimize code replication during CP code generation.

Inputs:

$\text{Refs}_{\text{read}}, \text{Refs}_{\text{write}}$	:	sets of read and write references in a single logical communication event
RefMap_r	:	map representing reference r, $\forall r \in \text{Refs}_{\text{read}} \cup \text{Refs}_{\text{write}}$
CPMap_r	:	computation partitioning map for reference r
Layout_A	:	layout of the common referenced array, denoted A
V_r	:	loop-level of innermost loop enclosing communication for r, after vectorization; $J_1 \ldots J_{V_r}$ are index variables of enclosing loops
$\underline{m}$	:	processor index vector for the representative processor m or *myid*

Algorithm:

(1) $\text{CPMap}_r^V = CPMap_r \bigcap_{range} \{[j_1, \ldots, j_n] : j_1 = J_1 \wedge \ldots j_{V_r} = J_{V_r} \wedge -\infty < j_{V_r+1} < \infty \wedge \ldots\}$

(2) $\text{DataAccessed}_t = \bigcup_{r \in \text{Refs}_t} \text{CPMap}_r^V \circ \text{RefMap}_r$, (hereafter, $t \in \{read, write\}$)

(3) $\text{NLDataAccessed}_t = \{ [\underline{p}] \rightarrow [\underline{a}] : \text{off-processor array elements } \underline{a} \text{ referenced by processor } \underline{p} \}$

$$= \begin{cases} \text{DataAccessed}_t - \text{Layout}_A & \text{if } t = read \\ \text{DataAccessed}_t \cap \text{Layout}_A & \text{if } t = write \end{cases}$$

$\text{nlDataSet}_t(\text{m}) = \text{NLDataAccessed}_t(\{\underline{m}\})$

(4) $\text{NLCommMap}_t(\text{m}) = \{ [\underline{p}] \rightarrow [\underline{a}] : \text{off-processor elements referenced by proc. } m \text{ and owned by proc. } \underline{p} \}$

$= \text{Layout}_A \bigcap_{range} \text{nlDataSet}_t(\text{m}))$

(5) $\text{LocalCommMap}_t(\text{m}) = \{ [\underline{p}] \rightarrow [\underline{a}] : \text{array elements owned by proc. } m \text{ to be communicated with proc. } \underline{p} \}$

$= \text{DataAccessed}_t \bigcap_{range} \text{Layout}_A(\{\underline{m}_A\})$

(6) $\text{SendCommMap(m)} = \text{LocalCommMap}_{\text{read}}(\text{m}) \bigcup \text{NLCommMap}_{\text{write}}(\text{m})$

(7) $\text{RecvCommMap(m)} = \text{NLCommMap}_{\text{read}}(\text{m}) \bigcup \text{LocalCommMap}_{\text{write}}(\text{m})$

Figure 3: Equations for computing communication sets

3.2 Implementing Explicit Communication

For message-passing systems, data-parallel compilers must compute the data to be exchanged between processors, and generate code to pack, communicate, unpack, and reference the non-local data. *Message vectorization* is a fundamental optimization for such systems which reduces communication frequency by hoisting communication for a reference out of one or more enclosing loops. To vectorize communication, the compiler must compute the set of data to send between each pair of processors; such communication sets depend on the reference, layout, and computation partitioning. *Message coalescing* combines messages for multiple references to eliminate redundant communication and further reduce the number of messages. Coalescing requires unioning communication sets and can require sophisticated techniques to enable efficient access to buffered non-local data.

Early phases in dHPF identify potentially non-local references that might access off-processor data, compute where to place communication for each reference (using dependence and optionally dataflow analysis to vectorize communication), and which sets of references can have their communication coalesced. dHPF assumes that all data is communicated to and from its owner(s) only, as defined by the data layout directives. A read reference is non-local if the location is not owned by the processor executing the read. A write reference is non-local if the location is owned by one or more processors besides the processor executing the write. (These definitions are equivalent if data is not replicated.) We refer to the sequence of messages required for a set of coalesced references as a single *logical communication event*.

Given sets of coalesced references and the placement level of communication, we compute the communication sets for each logical communication event using the inputs and set equations shown in Figure 3. The goal of these equations is to compute two maps, SendCommMap(m) and RecvCommMap(m), for the representative processor m = `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 two maps DataAccessed_t, $t \in \{read, write\}$, which specify the entire set of data accessed by each processor $\underline{p}$, via all read and write references in all iterations of the loops out of which communication has been vectorized. Then (step 3), the *non-local* data accessed by read references is the difference of DataAccessed_t and the local data owned by each processor, Layout_A. The non-local data accessed by write references is the intersection of DataAccessed_t and the data owned by any other processors. Note that the read and write equations are equivalent for the common case that each array element is owned by a single processor, i.e., where the data layout is not replicated.[2] We convert this map to a set $\text{nlDataSet}_t(\text{m})$ specifying the non-local data accessed by the fixed processor m.

We then compute two maps describing the non-local and local data (w.r.t. to the fixed processor m) that must be communicated with each other processor $\underline{p}$ (steps 4,5). Restricting the range of Layout_A to the non-local data set, $\text{nlDataSet}_t(\text{m})$, gives the non-local data referenced by pro-

[2] In this case, in fact, we can skip step (3) and simply use $NLDataAccessed_t = DataAccessed_t$, because steps (4) and (5) will restrict communication to non-local data. During code generation, we ensure that a processor does not communicate with itself.

cessor m and owned by each other processor p. Restricting the range of DataAccessed$_t$ to the local section owned by m gives the local data owned by m and accessed by each other processor p.

Finally, from LocalCommMap$_t$ and NLCommMap$_t$, $t \in \{read, write\}$, we compute the data to send to and receive from each processor (steps 6,7). Later, we generate code from these maps to pack and unpack the data at the sending and receiving ends.

The equations presented here unify the handling of both communication vectorization and coalescing for arbitrary references and communication patterns. This abstract formulation of static communication analysis has greatly simplified the core of the dHPF compiler and enabled efficient handling of general classes of computation partitionings and affine references.

3.3 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 the capability of generating code from integer sets.

FORTRAN arrays are stored in column-major order. Accordingly, a *rectangular* communication set C for data in an array A with n dimensions represents contiguous data if there exists a k such that for the 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 local index set of the array, and define $S_{<i>}$ to be the projection (i.e., range) of set S in dimension i, $1 \leq i \leq \text{rank}(S)$. Then the above condition can be formalized as:

$$\exists k \; s.t. \; 1 \leq k \leq n \;\; \wedge \;\; \bigwedge_{i=1}^{i=k} (C_{<i>} = A_{<i>}) \;\; \wedge$$

$$\text{IsConvex}(C_{<k>}) \;\; \wedge \;\; \bigwedge_{i=k+1}^{i=n} \text{IsSingleton}(C_{<i>})$$

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 (if it cannot be proved at compile-time). The predicate IsConvex(S) reduces to testing if the set ConvexHull$(S) - S$ is empty. 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 in-place communication cannot be proven at compile time, we synthesize code from the unproven predicates to repeat this scan and check at run-time, 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. 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.

There are currently two limitations of our implementation of this analysis in the dHPF compiler. First, we apply the compile-time test for in-place communication only to communication sets with only a single conjunct. Our compiler support can be generalized straightforwardly to handle disjoint disjunctions as well when the satisfiability conditions on all conjuncts are mutually exclusive. Second, the code generation for runtime evaluation of these predicates is currently incomplete.

3.4 Loop Splitting

Loop splitting (or non-local index set splitting) is a powerful but complex transformation that has been proposed to ameliorate two types of communication overhead: the cost of referencing buffered non-local data, and the latency of communication [21]. Both techniques involve splitting a loop to separate the iterations that access only local data from those that may access non-local data. First, buffer access costs arise when local and non-local data are stored separately, and the correct location must be chosen with a run-time check on each reference. After splitting local and non-local iterations, no checks are needed for references in the local iterations. Second, the latency of communication can be (partly) hidden by splitting because communication required for non-local iterations can be overlapped with computation of the local iterations.

The only implementation of this transformation we know of is in Kali [21], where the authors used set equations to explain the optimization but used case-based analysis to derive the iteration sets for special cases limited to one-dimensional distributions. This approach is only practical for a small number of special cases.

We extend the equations in [21] to apply to arbitrary sets of references, and any CP in our CP model. We first describe loop-splitting for communication overlap, because it subsumes splitting for buffer access. We apply loop-splitting to any perfect loop-nest (not necessarily innermost) containing at least one partitioned loop and having no dependences that prevent iteration reordering. Since, in dHPF, write references may be non-local, we split the set of iterations of such 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 as shown in Figure 4(a) for a loop-nest containing one or more potentially non-local references. (The equations shown are applied separately to each statement group in the loop nest.) The key steps are to compute localDataAccessed$_r$ (analogous to computing nlDataSet$_t$ in Figure 3), and then localIters$_r$ which are the iterations in which reference r accesses local data. The desired four sets are then computed by grouping localIters$_r$ by read and write references.

We schedule the communication and computation for this loop nest in the sequence shown in Figure 4(b). When NLRW is non-empty, we can overlap either read or write latency, but not both; a simple heuristic could be used to choose between the two. The sequence in the figure overlaps read latency with NLWOIters and LocalIters. When NLRW is empty, however, the latency for writes as well as reads can be overlapped with LocalIters by placing the SEND for non-local writes immediately after NLWOIters.

This form of splitting subsumes splitting for non-local buffer access. 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$_r$ = cpIterSet $-$ localIters$_r$, because the reference will access only non-local data in these iterations.

Code generation for loop splitting subsumes the operation of partitioning the loop by reducing the loop bounds, since each of the four loop sections is a subset of cpIterSet for

Inputs:

CPMap	:	Common CP map for each statement in given statement group SG
$\text{Refs}_{\text{read}}, \text{Refs}_{\text{write}}$	:	non-local read and write references in SG
RefMap_r	:	map representing reference r, $\forall r \in \text{Refs}_{\text{read}} \cup \text{Refs}_{\text{write}}$
Layout_A	:	layout of the common referenced array, denoted A
$\underline{m}_{CP}, \underline{m}_A$	:	processor index vectors as in Figure 3

Algorithm:

$$\text{cpIterSet} = \text{CPMap}(\{m\})$$

$$\text{dataAccessed}_r = \text{RefMap}_r(\text{cpIterSet})$$

$$\text{localDataAccessed}_r = \begin{cases} \text{dataAccessed}_r \cap \text{Layout}_A(\{\underline{m}_A\}) & \text{if t = read} \\ \text{dataAccessed}_r - \text{Layout}_A(\neg\{\underline{m}_A\}) & \text{if t = write} \end{cases}$$

$$\text{localIters}_r = \text{Ref}_r^{-1}(\text{localDataAccessed}_r)$$

$$\text{nlReadIters} = \text{cpIterSet} - \bigcap_{r \in \text{Refs}_{\text{read}}} \text{localIters}_r$$

$$\text{nlWriteIters} = \text{cpIterSet} - \bigcap_{r \in \text{Refs}_{\text{write}}} \text{localIters}_r$$

$$\text{localIters} = \text{cpIterSet} \cap \bigcap \text{localIters}_r$$

$$\text{nlRWIters} = \text{nlReadIters} \cap \text{nlWriteIters}$$

$$\text{nlROIters} = \text{nlReadIters} - \text{nlWriteIters}$$

$$\text{nlWOIters} = \text{nlWriteIters} - \text{nlReadIters}$$

(a) Computing local/nonlocal iteration sets

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

(b) Scheduling loop iterations

Figure 4: Loop splitting to overlap communication and computation.

the statement group. The code generation is performed as part of the hierarchical code generation framework for computation partitioning described briefly in Section 3.1 [1].

4 Extensions for Symbolic Distribution Parameters

A variety of set representations can be used to implement the set framework, with a wide range of expressiveness and efficiency. We observe that the primary benefit of using the integer set approach is that it enables simple but rigorous formulations of the key data-parallel optimization problems. This benefit is somewhat independent of the generality of the underlying set representation. The Omega library provides a powerful integer set representation based on FME, and the library has been invaluable in developing and prototyping the set-based formulations of the optimizations. Nevertheless, FME has two potential disadvantages which we address as follows.

First, algorithms based on FME have poor worst-case performance. A goal of our research has been to determine whether or not this general approach is practical for commercial HPF compilers. There is some previous evidence that poor cases may be unlikely to occur in practice [25]. Also, changing the formulation of a problem can be extremely effective in avoiding complex sets, and has improved running times by more than an order of magnitude in some cases. In practice, we have not found the cost of the set framework to be a problem so far, as shown in Section 6. Nevertheless, if the approach still proves impractical, we can use one of two alternatives (without sacrificing the benefits of the simple equational formulations). We can use a simpler set representation such as Data Access Descriptors [5], or we can use competitive algorithms that limit the time spent on any single optimization or code generation problem. Both approaches would fall back on runtime techniques (such as inspector-executor [27]) which are required in any case for irregular problems.

A second, fundamental limitation of Omega is that it does not permit symbolic coefficients in affine constraints, because multiplication of integer variables renders the underlying class of integer sets undecidable [10]. Symbolic multiplication is required to represent a symbolic stride, k, for example, $\{[i] : 1 \leq i \leq N \wedge \exists \alpha \; s.t. \; i = k\alpha + 1\}$. In compiling an HPF program, symbolic strides arise for any type of HPF distribution when the number of processors is unknown at compile time, for the $cyclic(k)$ distribution with unknown k, and for loops with unknown strides. We have extended our framework to accommodate the limitation on symbolic number of processors and k, 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, [19]).

4.1 An Optimized Virtual Processor Model

To circumvent the limitation on symbolic data distribution parameters, we use a standard technique that avoids representing these parameters explicitly within the set framework, and instead incorporates them directly during code generation [2, 13]. We refine this technique to provide signif-

Inputs: Same as in Figure 3
Algorithm:

$$\begin{aligned}
\text{busyVPSet}_t &= \bigcup_{r \in Refs_t} Domain(CPMap_r), \quad (\text{hereafter}, t \in \{read, write\}) \\
\text{allNLDataSet}_t &= \text{NLDataAccessed}_t(\text{busyVPSet}_t) \\
\text{vpsThatOwnNLData}_t &= \text{Layout}_{\text{Ar}}^{-1}(\text{allNLDataSet}_t) \\
\text{vpsThatAccessNLData}_t &= \text{Domain}(\text{NLDataAccessed}_t) \\
\text{activeSendVPSet} &= \text{vpsThatOwnNLData}_{\text{read}} \bigcup \text{vpsThatAccessNLData}_{\text{write}} \\
\text{activeRecvVPSet} &= \text{vpsThatAccessNLData}_{\text{read}} \bigcup \text{vpsThatOwnNLData}_{\text{write}}
\end{aligned}$$

(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 = PIVOT+1, 100
  do j = PIVOT+1, 100
    ! ON_HOME { A(i,j) }
    A(i,j) = ··· + A(PIVOT,j)
  enddo
enddo
```

(b) Gauss parallel loop in HPF

$$\begin{aligned}
\text{vpArray} &= \{[v_1, v_2] : 1 \le v_1, v_2 \le 100\} \\
\text{loop} &= \{[i,j] : \text{PIVOT} + 1 \le i,j \le 100\} \\
\text{CPMap} &= \{[v_1, v_2] \to [i,j] : i = v_1 \wedge j = v_2 \wedge \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{RefMap}_{\text{read}} &= \{[i,j] \to [\text{PIVOT}, j]\} \\
\text{RefMap}_{\text{write}} &= \emptyset(\text{no non-local writes}) \\
\text{busyVPSet}_{\text{read}} &= \{[v_1, v_2] : \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{NLDataAccessed}_{\text{read}} &= \{[v_1, v_2] \to [\text{PIVOT}, v_2] : \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{activeSendVPset} &= \{[v_1, v_2] : v_1 = \text{PIVOT} \wedge \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{activeRecvVPset} &= \text{busyVPSet}_{\text{read}}
\end{aligned}$$

(c) Active virtual processors in Gauss loop

Figure 5: Active virtual processors for computing, sending and receiving

icantly simpler code for *block* distributions (by far the most common in practice). We also add an additional optimization to to eliminate or reduce the runtime overhead in the resulting code. Gupta et al. [13] apply a similar strategy but their approach was based on detailed analysis of specific cases. Instead, we describe a general integer-set-based algorithm to perform this optimization. The VP model and optimization are as follows.

To avoid representing the distribution explicitly, we replace each physical processor array in our equations by a virtual processor (VP) array corresponding to using template indices (i.e., ignoring the distribute directive) in dimensions where the block size or number of processors is unknown, but using one virtual processor index per physical processor index in all other dimensions. (We do not require template sizes to be known constants.) We make this replacement simply by constructing the Layout mapping as a map from VP indices to data elements. All the analyses described in the previous sections then operate unchanged on this virtual processor domain. During code generation for each specific problem (e.g., generating a partitioned loop), we add extra enclosing loops that enumerate the VPs that are owned by the physical processor `myid`. Also, when generating code for communication, we must aggregate the messages to all the VPs belonging to the same physical processor. These code generation steps are described in Section 4.2.

The refinement we use for *block* distributions is as follows. Consider a template dimension of extent N that is *block*-distributed on a processor array dimension of extent P, and let $B = \lceil N/P \rceil$ be the block size. The precise representation of this distribution is $\{[t] \to [p] : Bp + 1 \le t \le Bp + B \wedge 1 \le t \le N \wedge 1 \le p \le P\}$. In this case, only the product term Bp is not directly representable. We compute a distribution onto virtual processors as follows: $\{[t] \to [v] : v \le t \le v + B - 1 \wedge 1 \le t \le N \wedge 1 \le v \le N\}$. Intuitively, this assumes that the virtual processor v "owns" template elements $[v, v+B-1]$. Each physical processor p is mapped to a unique virtual processor $v = Bp + 1$. However, any other value of v represents a fictitious virtual processor not mapped to any physical processor. The special physical processor id $\underline{m}$ used in Figures 3 and 4 is replaced by its virtual processor id $v_{m_i} = Bm_i + 1$. Therefore, the generated SPMD code is automatically parameterized by $\underline{v_m}$ instead of $\underline{m}$.

There are two key advantages in using this refinement for the *block* distribution, and both are due to the property that there is a single virtual processor per physical processor. First, this property implies that no additional virtual processor loops are required. In fact, $\underline{v_m}$ always represents a true physical processor (*myid*), and therefore a *block* distribution does not require any additional changes to the computational loops in the generated SPMD code (even with transformations such as non-local index set splitting). Second, the above property implies that the communication sets capture the entire section that must be communicated to each physical processor (for each *block* dimension). This enables some optimizations and simplifies code generation. For example, if all array dimensions are *block*-distributed, the equations in Section 3.3 can be applied to the communication set to determine whether in-place communication is feasible. For code generation, the only additional step required is to ensure that communication is not attempted with a fictitious virtual processor. This step is described in Section 4.2.

```
do v = vlb, vub
  ! pack data for v
  ! send data to v
enddo
```

(a) Initial SEND code from Domain(SendDataMap(m))

```
do firstVP = vlb, min(vub, vlb+P)
  pid = (firstVP-tlb) % P + tlb
  do v = firstVP, vp2, P
    ! pack data for v
  enddo
  ! send data to pid
enddo
```

(b) Separating physical and virtual processors

```
myVLB = ((activeSendLB-tLB)/P)*P + m + tLB
if (myVLB .lt. activeSendLB) myVLB = myVLB + P
do firstVP = vlb, min(vub, vlb+P)
  pid = (firstVP-tlb) % P + tlb
  do myVP = myVLB, activeSendUB, P
    do v1 = firstVP, vub, P
      ! pack data for v1
    enddo
  enddo
  ! send data to pid
enddo
```

(c) Adding active virtual processor loop (final SEND code)

Figure 6: Code generation for SEND with optimized virtual processor model

For *cyclic* and *cyclic*(k) distributions, there are multiple virtual processors for each physical processor. However, not all virtual processors owned by a physical processor are necessarily "active" in any particular operation (a partitioned computation, sending data, or receiving data). An additional optimization step can be used to eliminate or reduce runtime overhead by restricting the virtual processor loop to the set of VPs owned by processor *myid* that are actually active. Since such a set for a single physical processor would not be directly representable, in general, we do this in two steps. We first use the integer-set equations shown in Figure 5(a) to compute the set of active VPs *across all processors* for the problems of interest. Then, we generate a loop nest from this set and explicitly rewrite the loop bounds to restrict it to the active VPs owned by processor *myid*.

First, the set of active VPs for any partitioned computation, denoted busyVPSet, is simply the domain of CPMap. (As in Figure 4, this must be applied for each statement group in a loop nest.) Second, for each logical communication event, the active VPs that must send or receive data can be computed directly from NLDataAccessed_t, the map from processors to non-local data referenced by each processor ($t \in \{read, write\}$). This map is already computed for communication generation (Figure 3). The map is first used to compute the sets of all virtual processors that own non-local data and all those that access non-local data ($\text{vpsThatOwnNLData}_t$ and $\text{vpsThatAccessNLData}_t$). These in turn directly provide the virtual processors that must be active in sending or receiving data.

The results of these equations are illustrated for the Gaussian Elimination example in Figure 5(b,c), where the reference to the pivot row, $A(PIVOT, j)$, 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 (PIVOT) must send any data, but all VPs active in the computation (busyVPSet) must receive non-local data. In practice, we can generate code so that only one VP per physical processor will receive each such element.

4.2 Code Generation using Virtual Processors

Given the active virtual processor sets computed above, a few conceptually simple steps are required to generate final SPMD code.

For a computational loop nest, if the CPs of all statement groups are described via *block*-distributed arrays, no additional steps are required as explained earlier. If any array dimension used in a given CP has a *cyclic* or *cyclic*(k) distribution, we must enclose the loop nest with one extra loop for each such dimension. We generate these loops as follows. We first generate a loop nest to enumerate the elements of the set busyVPSet computed from the CP. For each loop in the nest, we then directly modify its bounds and stride to restrict the loop to the active VPs owned by processor m. For example, for the i dimension in Figure 5, the final VP loop would have $lb = (PIVOT/P) * P + m + 1 + (m < PIVOT\%P)?P : 0$, $ub = 100$, and $step = P1$. If non-local index-set splitting is applied to the loop nest, the same virtual processor loop nest would have to be wrapped around each loop nest section.

For communication code, we describe the steps required for the sending side; the receiving code is similar. The steps are illustrated in Figure 6, assuming a 1D processor array with P processors, a template with lower bound tLB, and a *cyclic* distribution. First, we generate a loop nest to enumerate Domain(SendDataMap(m)), viz., the virtual processors to which processor m must send data, and insert code to pack data for v by enumerating Range(SendDataMap(m)) (part (a) of the figure). Then, we rewrite this loop to separately enumerate the physical processors, and the virtual processors for each physical processor (part (b)). Finally, we enumerate the active sending processors owned by m, by applying the procedure described above (for busyVPSet) to the set activeSendVPSet. This loop nest is wrapped around the inner virtual processor (packing) loop, yielding the final send code in part (c). (The *block* distribution case is again much simpler because the two inner virtual processor loops are not required.)

More generally, if the original loop-nest of part (a) had r loops in a (possibly imperfect) loop nest, we generate r physical processor loops with the same nesting structure. If k of these loops correspond to *cyclic* data dimensions, we generate $2k$ virtual processor loops in a single perfect loop nest enclosing the packing code. Finally, we insert one copy of this virtual processor loop nest into each innermost physical processor loop.

The extensions described above allow us to use a very general set representation without unduly restricting the use of symbolic quantities. The additional complexity introduced by these techniques is largely encapsulated in the implementation of the framework, and (we believe) are outweighed by the analysis capabilities and flexibility the integer set framework provides.

5 Implementation Issues

In the course of implementing the set framework in the dHPF compiler, several interesting technical issues arose that are worth noting.

Handling multiple processor arrays in HPF. An HPF program may declare multiple processor arrays with different ranks or extents. It is not meaningful to combine sets of tuples from different processor arrays in a single operation. It is also not possible to convert all processor index tuples into a common (e.g., linear) domain because the conversion would require a symbolic multiplication if some of the processor array sizes were unknown. For example, the union operation used to compute CPMap in Section 3.1 cannot be performed directly when the arrays A_k are distributed on different processor arrays. The implications of this are as follows.

First, the $CPMap$ or any set or mapping derived therefrom is stored as a list of the sets or mappings, representing an implicit union or intersection of the list members. (Note that arbitrary CPs on the same processor domain can still be unioned explicitly into a single mapping.) The range of such a mapping list can be converted back into a single set without any loss of precision (e.g., CPMap($\{\underline{m}\}$), Range(SendDataMap) or Range(RecvDataMap) in Figure 3). The domain of such a mapping, however, cannot be combined into a single set, e.g., Domain(SendDataMap) specifying the set of processors to whom data should be sent. We account for such cases explicitly during code generation from such a set at the cost of some runtime overhead. We expect it to be rare in practice to have a CP from multiple different processor arrays.

Second, in Figure 3, the difference and intersection operations in Equation 3 cannot be performed directly since the domains of DataAccessed_t and Layout_A may represent different processor arrays. Instead, we convert the maps to sets of data for the fixed processor `myid` (represented by the singleton sets $\{\underline{m}_{CP}\}$ and $\{\underline{m}_A\}$ in the two domains), and then compute $nlDataSet_t(m)$ directly as follows:

$$(3)\quad \text{nlDataSet}_t(\text{m}) = \begin{cases} \text{DataAccessed}_t(\{\underline{\text{m}}_{\text{CP}}\}) - \text{Layout}_A(\{\underline{\text{m}}_A\}), & t = read \\ \text{DataAccessed}_t(\{\underline{\text{m}}_{\text{CP}}\}) \cap \text{Layout}_A(\neg\{\underline{\text{m}}_A\}), & t = write \end{cases}$$

Finally, $NLDataAccessedMap_t$ is also required in Figure 5 but in this case it is required as a map. In the rare case that we have CP terms from multiple processor arrays, we cannot compute the busyVPSet explicitly and so cannot optimize the runtime checks.

Minimizing intermediate set sizes. In our experience, the two primary causes of high compile-time in the set framework are set operations that produce a large number of disjunctions, and code generation operations from sets that cause excessive code replication. (The latter is discussed below.) Disjunctions are expensive because the cost of simplifying sets after a sequence of high-level operations increases quadratically with the number of disjunctive terms. For this reason, where possible, we avoid computing large disjunctions in intermediate sets.[3] The equations of both Figures 3 and 4 are specifically formulated for this purpose. In Figures 3, we use Equation (3) to combine the DataAccessed_r maps for all reads and all writes, rather than applying equations (4)-(7) for individual references and performing a union of potentially many terms at the end. The DataAccessed_r maps are inherently much simpler and therefore it is easier to eliminate excess disjunctive terms. In Figure 4, we first compute the intersection of local iterations, $\bigcap_r \text{localIters}_r$, and use that to derive nlReadIters and nlWriteIters. The more intuitive formulation we had originally used was to compute the non-local iterations for each reference and then take unions over all read and all write references. In statement groups with many references, this sometimes produced intermediate sets with many more disjunctive terms.

[3] We are grateful to Bill Pugh for several suggestions on how to control set complexity in practice.

Limiting Code Replication. We take two approaches to limit code replication. We factor loop invariant constraints from iteration spaces before code generation and we carefully limit guard lifting. We discuss these issues in turn below.

CPs with multiple disjunctive terms arise fairly frequently in practice, particularly for IF and DO statements which get the union of the CPs of statements control dependent on them. When a statement has a disjunctive CP, the MMCODEGEN algorithm first computes disjoint disjunctive form and then generates separate code for each of the resulting terms. When the disjunction arised from a loop-invariant condition, this code replication is unnecessary. To minimize code replication, we factor each of the iteration spaces in a compound statement into two parts, just before code generation: (1) the set of *iteration space* constraints that relate to the bounds of the iteration space, and (2) a set of *other constraints* that are loop-invariant and unrelated to these bounds. We then generate code separately for these two pieces. The latter provides a sequence of one or more disjoint IF statements which we fold together and then insert a single copy of the code for the iteration space within. As an additional optimization, we project away all enclosing loop variables from *other constraints* before generating guard code (since our CPs enforce these at outer loop levels).

In generating code for a non-convex iteration space, guard overhead is reduced by lifting guards out of loops, but this in turn can cause significant code duplication. The MMCODEGEN function enables the user to specify how many levels guards should be lifted out when generating code (see Appendix A). We generally lift guards 1 level when generating a code template for a (not necessarily innermost) perfect loop nest. However, to avoid excessive code replication we don't lift guards out of loops that contain communication , or out of fully replicated loops (such as time-step loops). Also we disallow code replication at the procedure scope, where MMCODEGEN would otherwise attempt to duplicate and regroup statements to exploit partial overlap in any guard conditions for the statements.

6 Compiler Performance

In this section, we assess the performance of the dHPF compiler in terms of the detailed costs of applying the set-based compilation techniques described in previous sections. The benchmarks we use are TOMCATV—a SPEC92 benchmark that performs mesh generation, and SP—a serial version of one of the NAS parallel application benchmarks. Both programs perform stencil-based computations on multidimensional arrays.

The version of TOMCATV we studied is simply the Fortran 77 code for the SPEC92 benchmark with HPF directives to specify a (`BLOCK, *`) distribution of the arrays over a 1D processor grid. With our directives added, this benchmark

Breakdown of compilation time			
application	SP-4	SP-sym	T-sym
total compilation wall-clock time	1145s	1073s	28s
interprocedural analysis	1.2%	1.4%	1.5%
module compilation	97.9%	97.8%	97.1%
partitioning computation	14.5%	11.3%	16.1%
loop splitting	6.4%	2.0%	3.7%
loop bounds reduction	5.6%	6.7%	6.1%
communication generation	31.4%	34.6%	28.1%
loops to compute msg sizes	12.9%	13.4%	7.0%
loops over comm partners	12.8%	14.1%	10.4%
check if msg is contiguous	1.3%	1.9%	2.6%
check if msg is rect section	1.2%	1.4%	1.4%
opt of generated code	28.1%	28.9%	21.3%
mult mappings code generation	26.4%	23.9%	10.5%

Table 1: Breakdown of dHPF compilation time.

has 228 lines and a single procedure. Compared to TOMCATV, the SP application benchmark is more than an order of magnitude larger and the computation is considerably more complex. SP has much larger and non-uniform loop nests, procedure calls within parallel loops, and makes liberal use of privatizable arrays whereas TOMCATV uses only privatizable scalars. We developed an HPF version of SP using minimal modifications to a serial Fortran 77 code from the NPB2.3-serial release[4]. We specified *block* distributions in the y and z spatial dimensions of the program's 3D and 4D arrays. Our modified version of the source is 3502 lines, compared to the 3382 lines in the NPB2.3-serial release. The application has 30 procedures.

We used a version of dHPF compiled with `-O2` optimization and measured compile times on a 250MHz UltraSparc workstation. We used Rational Software's Quantify™ utility to obtain an execution profile. For TOMCATV, the number of processors was left unspecified at compile-time. For SP, we considered two variants: SP-4 used a fixed 2x2 processor array, while SP-sym left the total number unspecified, using a `2 x (number_of_processors()/2)` processor array. For both benchmarks, the compiler exploited all of the optimizations described in previous sections, including non-owner-computes computation partitionings for some statements to reduce the number and frequency of communication operations.

Table 1 shows the wall-clock time in seconds for compiling each of these benchmark versions and a breakdown of total execution time spent in key phases of dHPF, including each of the major integer-set optimizations. (The final optimization of generated code does not use integer-set-based operations.) The nesting of phases is shown by indentation in the first column. The numbers do not sum to 100% because percentages shown for indented phases are merely refinements of their enclosing phase. Although the benchmarks are quite different in size and complexity, the breakdown of compilation time for them is remarkably consistent. None of the phases is especially dominant in compile time, although SP has a somewhat high cost for communication generation because of a huge number number of communications (even after a high degree of coalescing), some with complex patterns. Based on these results, we would anticipate that other programs would have a similar cost breakdown although the distribution of compilation effort would differ depending on the number and complexity of loops in a routine, the number of communication events, and the shape of the communication sets.

At the bottom of the table, we note the time spent in Kelly, Pugh and Rosser's multiple mappings code generation operation which we use to synthesize loops that enumerate iteration or data sets. This algorithm accounts for virtually all of the cost of the set framework. Most of this time is spent in simplifying Presburger formulae representing integer sets and mappings. These numbers show that the integer-set framework is not a dominant cost in compile-time, even for fairly large and complex codes such as SP. Finally, the table shows that there is no significant additional cost to compiling for a symbolic number of processors vs. a known (fixed) number. SP-sym is in fact faster than SP-4 because the compiler performs more aggressive loop-splitting in the latter, which leads to more complex sets and correspondingly higher compile-time cost.

The absolute compile-times for SP are somewhat high but (we believe) acceptable for a research compiler where efficiency in the implementation has not been a primary goal. There are opportunities for significant improvements. Approximately 30% of compilation time is spent on generating custom inline code for counting, packing and unpacking buffers. While such custom code can be important for complex communication patterns, for the vast majority of simple patterns such as a shift communications, invoking a run-time library operation would not be more expensive, and would largely eliminate the communication generation cost. We also spend nearly 30% in a post-pass optimizing the SPMD code we generate, which we believe can be largely eliminated through an algorithmic improvement.

7 Preliminary Results

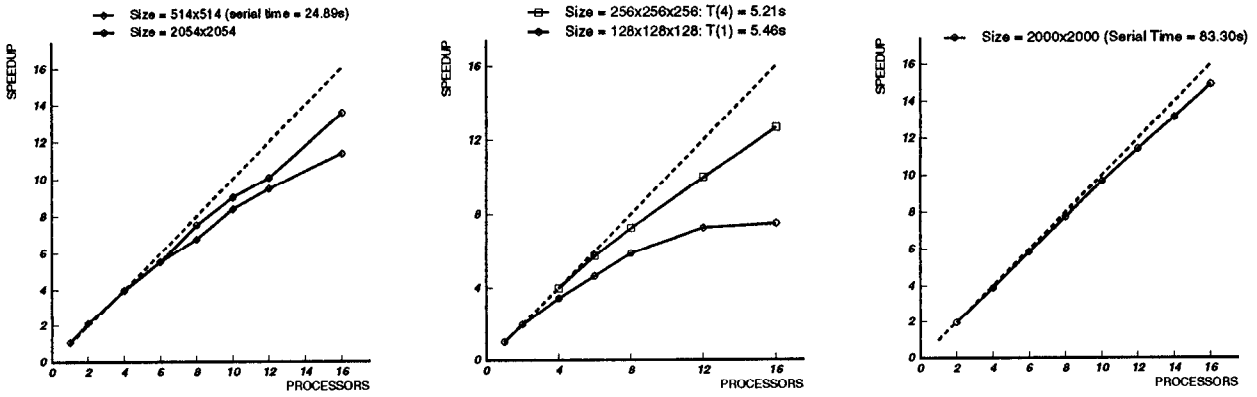
The purpose of this section is primarily to demonstrate that our compiler implementation using an integer set framework is practical and effective enough to achieve good speedups for realistic kernels and benchmarks. Many of the specific optimizations we have described in this paper are not new, although few if any compilers have implemented more than a few of these in a general way. Evaluating the impact of particular optimizations on applications is not directly relevant to this paper but is a subject of our current work.

We present results on the overall performance of three benchmark codes compiled with dHPF. The codes are TOMCATV, ERLEBACHER–a larger 3D compact differencing code, and JACOBI—a simple 4-point stencil kernel with a convergence loop. These results are preliminary and a more extensive performance evaluation with larger codes (including the NAS benchmarks) and comparisons with other compilers is currently under way.

We distributed the data arrays (`BLOCK,*`) in TOMCATV, (`*,*,BLOCK`) in ERLEBACHER, and (`BLOCK,BLOCK`) in JACOBI. The number of processors was left unspecified in each case. For JACOBI, we used a 2D processor array of shape (`2, number_of_processors()/2`), while the other benchmarks used 1D processor arrays. The Fortran77+MPI code generated by dHPF was compiled using the IBM xlf compiler at optimization level -O2. It was executed on an IBM SP-2 using the IBM MPI library with the user-level communication layer.

Figure 7 shows the resulting speedups for the three codes. Speedups for JACOBI and the small problem sizes for ERLEBACHER and TOMCATV are relative to the original serial version of each benchmark. The larger problem sizes for ERLEBACHER and TOMCATV exceed the available physical memory on smaller numbers of processors. For this reason,

[4]The NAS benchmarks are available from `http://science.nas.nasa.gov/Software/NPB`.



(a) Speedups for TOMCATV (b) Speedups for ERLEBACHER (c) Speedups for JACOBI

Figure 7: Speedups for benchmarks on an IBM SP2.

speedups of the large problem size for TOMCATV and ERLEBACHER are computed relative to the 4-processor speedup as $4*T(4)/T(p)$. For TOMCATV, the compiler provides moderate speedups for a small problem size of 514x514. For the larger problem, we see that the scaling improves. The reduced scalability on smaller problems is primarily because of two global `maxloc` reductions within a relatively small main computational loop. (The compiler recognizes and implements these reductions extremely efficiently [23].) 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. The scalar efficiency of our generated code is quite good. For TOMCATV, our generated code actually achieves a speedup of 1.08% on one processor over the original, perhaps because of a reduction in cache conflicts that is a side effect of our changes to data layout.

Measurements of TOMCATV showed some interesting benefits from loop-splitting for efficient buffer access. In TOMCATV, data cannot be received in-place with the (BLOCK, *) distribution, and for such stencil computations compilers have traditionally unpacked the received data into overlap areas. However, the loop-splitting into local and non-local iterations enabled us to reference data directly from the receive buffer *without* unpacking. For the smaller problem size, this actually proved faster than when using overlap areas [24]. For the larger problem size, the two versions performed approximately as well. Loop-splitting is essential to achieving such good performance without overlap areas. This result is significant because overlap areas only apply to nearest-neighbor communication patterns whereas loop splitting is applicable to arbitrary patterns.

For ERLEBACHER, several factors limit the speedup: a pipelined communication pattern with numerous relatively small messages, a broadcast communication of a large panel of data, a large reduction of a 3D to a 2D array, and a substantial load imbalance in computing boundary conditions. Nevertheless, the principal computation phases are efficiently parallelized by dHPF, and the compiler is able to achieve fairly good scaling in performance for the larger problem size. In addition to static computation partitioning, message vectorization, and simple instances of message coalescing, the compiler again applied loop splitting for both communication-computation overlap and efficient buffer access. Loop-splitting permitted the code to reference nonlocal data directly from receive buffers for the large broadcast messages, where overlap areas were not useful.

For JACOBI, the speedup scales linearly as should be expected for this simple, regular stencil code. The optimizations applied here included static loop partitioning identical to the owner-computes rule, using in-place send and receive operations along one of the two dimensions, and recognizing the reduction for the convergence loop.

Currently, we are experimenting with the NAS application benchmarks, LU, BT, and SP (as described on the previous section). We have successfully compiled versions of these benchmarks with dHPF, and are currently experimenting with advanced optimizations that leverage our integer set framework. Optimizations we have designed and implemented in dHPF for handling such real-world codes include interprocedural computation partitioning, effective computation partitioning for privatizable arrays (which uses partial replication of computation to reduce communication), and communication-sensitive loop distribution. Preliminary results have been encouraging and we have measured better than of 85% efficiency on 4 processors for BT, as well as good scaling behavior up to 16 processors. We expect to use these codes to compare the performance of dHPF with other selected compilers, and to evaluate the benefit of the specific optimizations implemented in dHPF.

8 Related Work

As explained in the Introduction, most research and commercial data-parallel compilers to date use case-based approaches for implementing basic communication and iteration set analysis [22, 7, 21, 29, 11, 12, 14, 8, 13, 31, 32, 33] This is a fundamentally different approach from that taken in this paper, and its strengths and weaknesses have been discussed in Section 1.

Previously, Hiranandani et al. [16] and Koelbel [21] both formulated the basic computation partitionings and communication analysis problems in terms of abstract integer-set operations, but their formulations assumed the owner-computes rule. Our formulations apply to a much more general partitioning model, and support more general communication coalescing, non-local index set splitting, and in-place communication analysis. Furthermore, they used case-based approaches in their implementations of the Kali and Fortran D compilers. In contrast, we present an actual and fairly general implementation in terms of integer set operations, address a fundamental limitation in the representation

of symbolics, and demonstrate experimentally that such an implementation is practical for large, real codes.

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., [13, 9, 19, 29, 32]). These techniques would provide more efficient support for *cyclic*(k) distributions but would be much less efficient if used in the general form for *block* and *cyclic* distributions.

Previous work on using linear inequalities and Fourier-Motzkin Elimination [28] for code generation shares our goal of improving the generality, the level of abstraction, and the quality of code in data-parallel compilers. Three groups (Ancourt et al. [3], the Paradigm compiler group [30] and SUIF [2]) applied these techniques to support code generation for communication and iteration sets described by linear inequalities. The latter work also describes how basic communication analyses and optimizations such as message coalescing and redundant communication elimination can be performed using linear inequalities. However, since these groups did not use a representation capable of representing general non-convex sets, it precluded performing potentially important optimizations such as precise communication coalescing for arbitrary affine references, splitting iterations sets into local and non-local components, or the use of a more general computation partitioning model. All of these are possible in dHPF, and in fact have been used for many or all of our benchmarks. Furthermore, all of our optimizations fully support our general computation partitioning model.

Our virtual processor approach for capturing distributions on symbolic numbers of processors is similar to the approach used by Gupta et al. [13] for computing communication sets and that used by SUIF [2] for *cyclic* distributions. As discussed in Section 4, compared to SUIF, the key improvement in our work and the work of Gupta et al. is to compute active processor sets and use these to eliminate or reduce runtime checks in the generated code. The primary difference in our algorithm compared to Gupta et al. is that they compute the active VP sets by detailed analysis of specific cases whereas we describe a simpler, more general algorithm expressed as integer set equations.

Perhaps the most important distinction between our work and previous data parallel compilers is that (we believe) we are the first to demonstrate by implementation experience that it is practical to build a complete compiler for High Performance Fortran using a general and powerful integer-set framework.

9 Conclusions

We have described an integer-set-based approach for analysis and code generation for data-parallel programs. We have used this framework as the basis for implementing very general versions of a comprehensive collection of data-parallel optimizations in the dHPF compiler. It is particularly significant that the framework has been able to support such comprehensive optimizations despite the fact that dHPF supports a more general computation partitioning model than has been implemented previously. Furthermore, the compiler has relatively modest compile times, and in particular, the analysis and code generation operations using the set framework account for only about 25% of the overall execution time of the compiler for the larger examples we studied.

Overall, our integer-set-based framework provides a simple, uniform, and powerful foundation for analysis, optimization and code generation of data-parallel programs. In dHPF, our implementation preserves much of the simplicity and expressiveness of the integer set framework while accommodating performance tuning to minimize the runtime overhead for important cases. Preliminary measurements show that our set-based analyses and optimizations enable us to achieve good performance on small codes. In our current work with larger programs, we have found that our uniform approach to analysis and optimization has facilitated rapid development of several sophisticated new optimizations that appear important for achieving high performance on these more complex codes.

Acknowledgments

Ajay Sethi collaborated in the development and implementation of early versions of the equational framework for communication sets. Guohua Jin provided invaluable assistance in obtaining the results for the NAS benchmark BT. The performance results we present in this paper would not have been possible without the efforts of individuals who contributed to the design and implementation of the dHPF compiler including Arun Chauhan, Chen Ding, Kathi Fletcher, Robert Fowler, Guohua Jin, Collin McCurdy, Mike Paleczny, Dejan Mircevski, Nenad Nedeljkovic, Monika Mevencamp, Bo Lu, and Ajay Sethi. Ken Kennedy provided invaluable guidance and leadership to the dHPF project.

We are grateful to Bill Pugh and Evan Rosser who provided valuable advice and technical assistance in using the Omega library effectively. Sarita Adve, Robert Fowler and Ken Kennedy provided valuable comments on earlier drafts of this paper.

References

[1] V. Adve and J. Mellor-Crummey. Advanced code generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, Lecture Notes in Computer Science Series. Springer-Verlag. (to appear).

[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, Dec. 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, Apr. 1991.

[5] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[6] 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, Oct. 1995.

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

[8] 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, Feb. 1995.

[9] 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, Dec. 1992.

[10] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[11] 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.

[12] 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, Dec. 1995.

[13] S. K. S. Gupta, S. D. Kaushik, C. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, Feb. 1996.

[14] 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.

[15] 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.

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

[17] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.

[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, Feb. 1995.

[19] 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.

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

[21] 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, Oct. 1991.

[22] 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.

[23] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, Mar. 1998. To appear.

[24] J. Mellor-Crummey and V. Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 1998. (to appear). A previous version of this paper was presented at the 1997 Workshop on Languages and Compilers for Parallel Computing.

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

[26] 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.

[27] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, Apr. 1990.

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

[29] 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, Apr. 1994.

[30] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. H. 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.

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

[32] K. van Reeuwijk, W. Denissen, H. Sips, and E. 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, Sept. 1996.

[33] 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

Following the notation used by the Omega library[17], we use the following sets and maps to define some of the less common operations used in the paper:

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

Some of the key operations that we use in our equational framework are defined as follows:

Composition of two maps :

$$R1 \circ R2 \equiv \Big\{ [i_1 \ldots i_n] \rightarrow [j_1 \ldots j_p] : \exists \alpha_1 \ldots \alpha_m \; s.t. \; f_{R1}(i_1 \ldots i_n, \alpha_1 \ldots \alpha_m) \wedge f_{R2}(\alpha_1 \ldots \alpha_m, j_1 \ldots j_p) \Big\}$$

Composition of a map with a set :

$$R1(S1) \equiv \Big\{ [j_1 \ldots j_m] : \exists \alpha_1 \ldots \alpha_n \; s.t. \; f_{R1}(\alpha_1 \ldots \alpha_n, j_1 \ldots j_m) \wedge f_{S1}(\alpha_1 \ldots \alpha_n) \Big\}$$

Restrict Range :

$$R1 \bigcap_{\text{range}} S2 \equiv \Big\{ [i_1 \ldots i_n] \rightarrow [j_1 \ldots j_m] : f_{R1}(i_1 \ldots i_n, j_1 \ldots j_m) \wedge f_{S2}(j_1 \ldots j_m) \Big\}$$

Codegen($S1 \ldots Sv$ | Known) :
$S1 \ldots Sv$ specify the iteration spaces for v statements. $Known$ is a rank 0 set of constraints on global variables in $S1 \ldots Sv$ that are guaranteed true.
CodeGen synthesizes a code fragment to enumerate the tuples in $S1 \ldots Sv$ in lexicographic order, where the same tuple in different sets is ordered as: $(\underline{i} \in Sj) \prec (\underline{i} \in Sk), j < k$.