Journal of Instruction-Level Parallelism 5 (2003) 1-29 Submitted 10/02; published 4/03

An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications*

Daniel Chavarría-Miranda DANICH@CS.RICE.EDU
John Mellor-Crummey JOHNMC@CS.RICE.EDU
Dept. of Computer Science
Rice University
Houston, TX

Abstract

Data parallel compilers have long aimed to equal the performance of carefully hand-optimized parallel codes. For tightly-coupled applications based on line sweeps, this goal has been particularly elusive. In the Rice dHPF compiler, we have developed a wide spectrum of optimizations that enable us to closely approach hand-coded performance for tightly-coupled line sweep applications including the NAS SP and BT benchmark codes. From lightly-modified copies of standard serial versions of these benchmarks, dHPF generates MPI-based parallel code that is within 4% of the performance of the hand-crafted MPI implementations of these codes for a 102^3 problem size (Class B) on 64 processors. We describe and quantitatively evaluate the impact of partitioning, communication and memory hierarchy optimizations implemented by dHPF that enable us to approach hand-coded performance with compiler-generated parallel code.

1. Introduction

A significant obstacle to the acceptance of data-parallel languages such as High Performance Fortran [1] has been that compilers for such languages do not routinely generate code that delivers performance comparable to that of carefully-tuned, hand-coded parallel implementations. This has been especially true for tightly-coupled applications, which require communication *within* computational loops over distributed data dimensions [2]. Loosely synchronous applications, which only require communication *between* loop nests, can achieve good performance without sophisticated optimization. Without precise programmer control over communication and computation or sophisticated compiler optimization, it is impossible for programmers using data-parallel languages to approach the performance of hand-coded parallel implementations with any amount of source-level tuning. Since high performance is the principal motivation for developers of parallel implementations, scientists have preferred to hand-craft parallel applications using lower level programming models such as MPI [3], which can be painstakingly hand tuned to deliver the necessary level of performance.

The Rice dHPF compiler project [4, 5, 6] has focused on developing data-parallel compilation technology to enable generation of code whose performance is competitive with hand-coded parallel programs written using lower-level programming models. The dHPF compiler supports compilation of applications written in a Fortran 77 style augmented with HPF data distribution directives.

*. This work has been supported by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23, as part of the prime contract (W-7405-ENG-36) between the Department of Energy and the Regents of the University of California.

©2003 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

dHPF supports a broad spectrum of novel data-parallel optimizations that enable it to produce high-quality parallel code for complex applications written in a natural style.

This paper uses the NAS SP and BT computational fluid dynamics application benchmarks [7] as a basis for a detailed performance study of parallel code generated by dHPF. Both the SP and BT codes are tightly-coupled applications based on line sweeps. Line sweep computations are used to solve one-dimensional recurrences along a dimension of a multi-dimensional hyper-rectangular domain. This strategy is at the heart of a variety of numerical methods and solution techniques [8]. SP and BT employ line sweeps to perform Alternating Direction Implicit (ADI) integration—a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation [8, 9, 10, 11].

To evaluate the effectiveness of the dHPF compiler for parallelizing line-sweep computations, we compare the performance of several versions of the SP and BT codes parallelized by dHPF to other compiler-based and hand-coded parallelizations. These comparisons show that dHPF generates parameterized parallel code with scalable performance that closely approaches that of the hand-coded parallelizations by NASA. The remainder of the paper focuses on a quantitative evaluation of the performance contribution of dHPF's key optimizations. We determine how each key optimization contributes to the performance of dHPF's generated code by comparing overall application performance with and without that optimization.

Section 2 provides some brief background about the dHPF compiler and the NAS SP and BT benchmarks. Section 3 provides an overall comparison of the performance of dHPF-generated code against hand-coded and other compiler-based parallelizations of SP and BT. Section 4 compares the performance impact of several different compiler-based partitioning strategies suitable for SP and BT. Section 5 describes and evaluates communication optimizations in dHPF. Section 6 describes and evaluates optimizations to improve memory hierarchy utilization. Section 7 presents our conclusions and outlines open issues.

2. Background

2.1 The dHPF Compiler

The Rice dHPF compiler has several unique features and capabilities that distinguish it from other HPF compilers. First, it uses an abstract equational framework that enables much of its program analysis, optimization and code generation to be expressed as operations on symbolic sets of integer tuples [5]. Because of the generality of this formulation, it has been possible to implement a comprehensive collection of advanced optimizations that are broadly applicable.

Second, dHPF supports a more general computation partitioning model than other HPF compilers. With few exceptions, HPF compilers use simple variants of the *owner-computes* rule [12] to partition computation among processors: partitioning each statement's instance (or alternatively, each loop iteration [13]) is based on a single (generally affine) reference. Unlike other compilers, dHPF permits independent computation partitionings for each program statement. The partitioning for a statement maps computation of its instances onto the processors that own data accessed by one (or more) of a *set* of references. This computation decomposition model enables dHPF to support sophisticated partially-replicated partitionings; such partitionings have proven necessary for achieving high performance with the NAS SP and BT benchmarks, as we describe in later sections.

Finally, the dHPF compiler provides novel compiler support for multipartitioning [14, 15], a family of sophisticated skewed-cyclic block distributions that were originally developed for hand-coded parallelization of tightly-coupled multi-dimensional line-sweep computations [8, 9, 11]. Multipartitioning enables decomposition of arrays of $d \geq 2$ dimensions among a set of processors so that for a line sweep computation along any dimension of an array, all processors are active in each step of the computation, load-balance is nearly perfect, and only coarse-grain communication is needed. We have extended HPF's standard data distribution directives with a new keyword, `MULTI`, that specifies a multipartitioned distribution. For a data distribution partitioned with `MULTI`, the `MULTI` keyword must be used in at least two dimensions; our implementation currently requires that all dimensions not partitioned with `MULTI` be unpartitioned.

2.2 NAS SP and BT Benchmark Codes

The NAS SP and BT application benchmarks [7] are tightly-coupled computational fluid dynamics codes that use line-sweep computations to perform Alternating Direction Implicit (ADI) integration to solve discretized versions of the Navier-Stokes equation in three dimensions. SP solves scalar penta-diagonal systems, while BT solves block-tridiagonal systems. The codes both perform an iterative computation. In each time step, the codes have a loosely synchronous phase followed by tightly-coupled bi-directional sweeps along each of the three spatial dimensions. These codes have been widely used to evaluate the performance of parallel systems. Sophisticated hand-coded parallelizations of these codes developed by NASA provide a yardstick for evaluating the quality of code produced by parallelizing compilers.

The NAS 2.3-serial versions of the SP and BT benchmarks consist of more than 3500 lines of Fortran 77 sequential code (including comments). To these, we added HPF data layout directives which account for 2% of the line count. To prepare these codes for use with dHPF, we manually inlined several procedures as we describe below. For SP, we inlined procedures `lhsx` and `ninvr` into the source code for procedure `x_solve`; `lhsy` and `pinvr` into `y_solve`; as well as `lhsz` and `tzetar` into `z_solve`. For BT, we inlined `lhsx`, `lhsy`, and `lhsz` into `x_solve`, `y_solve`, and `z_solve` respectively. The purpose of this inlining in SP and BT was to enable the dHPF compiler to globally restructure the local computation of these inlined routines so that it could be overlapped with line-sweep communication. The inlining was necessary to remove a structural difference between the hand-coded MPI and serial versions that would have required interprocedural loop fusion to eliminate automatically. In BT, one additional small inlining step was needed to expose interprocedurally carried dependences in the sweep computation to avoid having to place communication within the called routine.

2.3 Experimental Framework

All experiments reported in this paper were performed on a dedicated SGI Origin 2000 parallel computer with 128 R10000 250MHz processors. Each processor possesses 4MB of L2 cache, 32KB of L1 data cache and 32KB of L1 instruction cache. The Fortran code generated by source-to-source compilation with dHPF, along with the reference hand-coded versions of the NAS SP and BT benchmarks (version 2.3), was compiled using the SGI MIPSpro compilers and linked with SGI's native MPI library. Detailed metrics, such as cache misses and MPI operations were measured using hardware performance counters under control of SGI's `ssrun` utility. In our experiments, we measured SP and BT executions using both class A (64^3) and class B (102^3) problem sizes. To

see the impact of individual optimizations on scalability, we measured appropriate metrics for 16 and 64 processors. Metrics were collected for executions with only 10% of the standard number of iterations to keep execution time and trace file size manageable. Comparing performance metrics for 16 and 64 processors helps show differences in optimization impact for small and large data and processor sizes.

3. Performance Comparison

In this section we compare the resulting performance of *four* different versions of the NAS SP and BT benchmarks[1]:

- NAS SP & BT Fortran 77 MPI hand-coded version, implemented by the NASA Ames Research Laboratory,
- NAS SP & BT (multipartitioned) compiled with dHPF from sequential sources,
- NAS SP & BT (2D block) compiled with dHPF from sequential sources, and
- NAS SP & BT (1D block, transpose) compiled with the Portland Group's *pghpf* from their HPF sources.

The Portland Group's (PGI) versions of the NAS SP and BT benchmarks were obtained directly from the PGI WWW site [16].

PGI's HPF versions of these codes use a 1D BLOCK data distribution and perform full transposes of the principal arrays between the sweep phases of the computation. Since their compiler does not support array redistribution, their implementation uses *two* copies of two large 4D arrays, where the copies are related by a transpose. The PGI versions of the NAS SP and BT computations were written from scratch to avoid the limitations of the PGI compiler. A discussion of this topic can be found elsewhere [4].

We executed these code versions on 1–64 processors for the class 'A' (64^3) problem size and 1–81 processors for the larger class 'B' (102^3) problem size. This range of problem sizes and processor counts is intended to show the performance of different variants of the benchmarks for a range of per-processor data sizes and communication-to-computation ratios.

Figure 1 compares the efficiency of four different parallelizations of the NAS SP benchmark for the class 'A' and 'B' problem sizes respectively. Figure 2 presents the efficiency comparisons for the NAS BT benchmark's class 'A' and 'B' problem sizes.

For each parallelization ρ, the efficiency metric is computed as $\frac{t_s}{P \times t_p(P,\rho)}$. In this equation, t_s is the execution time of the original sequential version implemented by the NAS group at the NASA Ames Research Laboratory; P is the number of processors; $t_p(P,\rho)$ is the time for the parallel execution on P processors using parallelization ρ. Using this metric, perfect speedup would appear as a horizontal line at efficiency 1.0. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gauge the relative performance of multiple versions across the *entire* range of processor counts.

The graphs show that the efficiency of the hand-coded MPI-based parallelizations based on a multipartitioned data distribution is excellent, yielding an average parallel efficiency of 1.20 for SP

1. The source and generated code used for our experiments is available at `www.cs.rice.edu/~dsystem/dhpf/pact2002`.

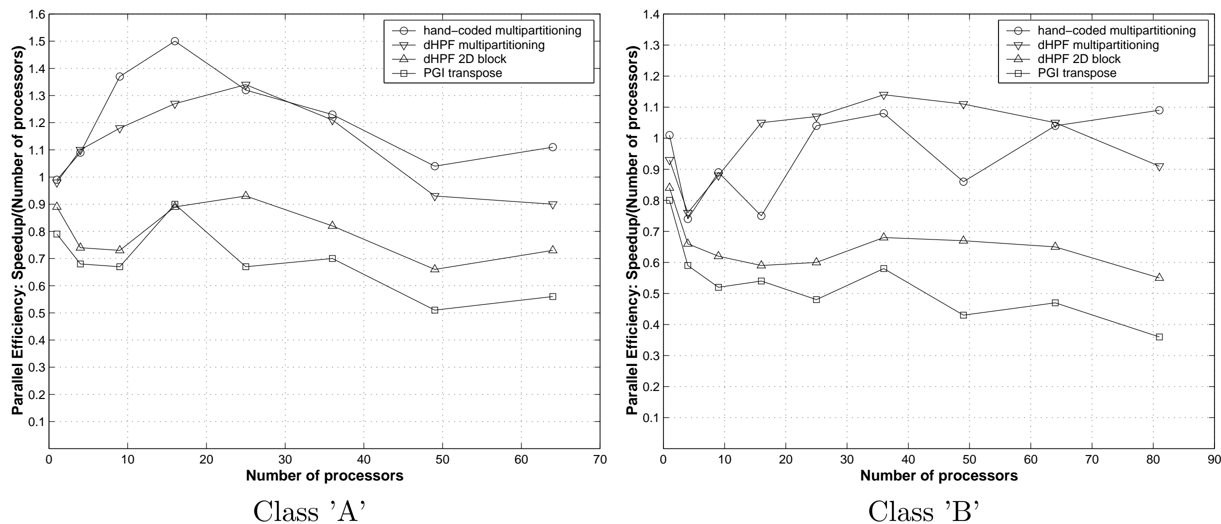


Figure 1: Parallel efficiency for NAS SP

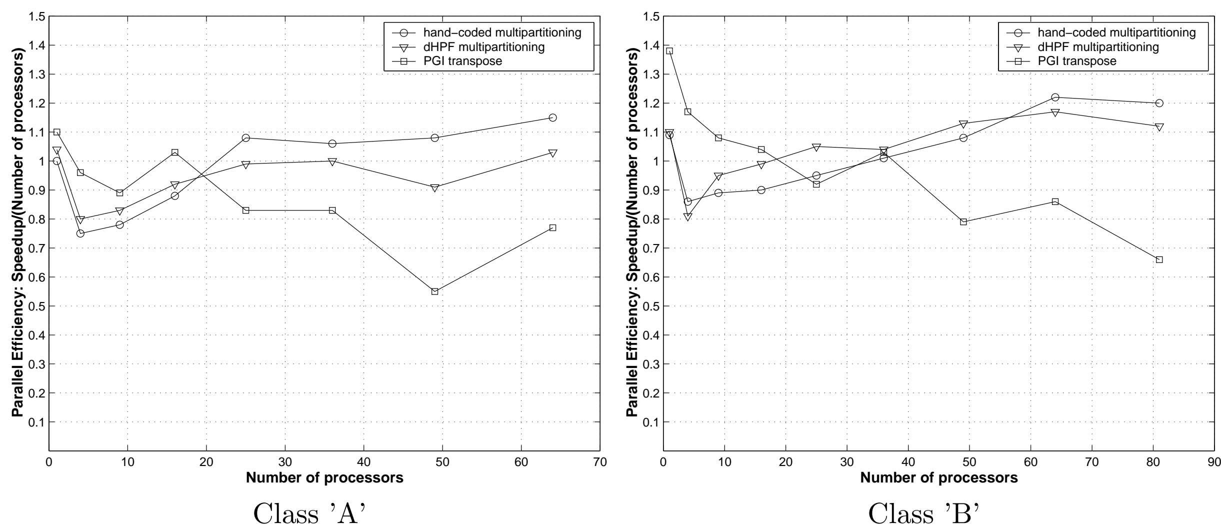


Figure 2: Parallel efficiency for NAS BT

class 'A', 0.94 for SP class 'B', 0.97 for BT class 'A' and 1.02 for BT class 'B'. Thus, the hand-coded versions achieve roughly linear speedup.

The dHPF-generated multipartitioned code achieves similar parallel efficiency and near-linear speedup for most processor counts, demonstrating the effectiveness of our compilation and optimization technologies. Its average efficiency across the range of processors is 1.11 for SP class 'A', 0.99 for SP class 'B', 0.94 for BT class 'A' and 1.04 for BT class 'B'. The dHPF compiler achieves this level of performance for code generated for a symbolic number of processors, whereas the hand-coded MPI implementations have the number of processors compiled directly into them.

The remaining gaps between the performance of the dHPF-generated code and that of the hand-coded MPI can be attributed to two factors. First, the dHPF-generated code has a higher secondary cache miss rate on large numbers of processors due to interference between code and

data in the unified L2 cache of the Origin 2000. The memory footprint of dHPF's generated code is substantially larger than that of the hand-coded version due to its generality. Second, on 81 processors the class 'B' benchmarks suffer from load imbalance on tiles along some of the surfaces of multipartitioned arrays. dHPF uses a fixed block size (namely, $\lceil \frac{s}{t} \rceil$, where s is the extent of the dimension and t is the number of tiles in that dimension) for tiles along a dimension; using this scheme, one or more tiles at the right edge of a dimension may have fewer elements. For the class 'B' benchmarks on 81 processors, this leads to partially-imbalanced computation with 8 tiles of size 12 and a tile of size 6 at the right boundary. In contrast, the tiling used by the hand-coded MPI ensures each processor has either $\lfloor \frac{s}{t} \rfloor$ or $\lceil \frac{s}{t} \rceil$ elements, which leads to more even load balance.

The performance of the dHPF-generated 2D block partitioning using coarse-grain pipelining (CGP) [4] is respectable, with an average efficiency of 0.80 for SP class 'A' and 0.65 for class 'B'. Due to a compiler limitation, the CGP version of SP uses overlap regions for storing off-processor data; this places it at a disadvantage with respect to the multipartitioned version, which accesses off-processors data directly out of communication buffers. Even if this limitation were eliminated, it would not boost the performance to the level achieved by the multipartitioned codes; we discuss this issue in more detail in the next section. Due to a limitation in our dependence analysis, we were not able to obtain results for BT using this partitioning scheme.

Figures 1 and 2 also show the efficiency of the version of the NAS SP and BT benchmarks written by PGI, which use 1D block distributions with transposes between sweep phases. These codes, compiled with version 2.4 of the *pghpf* compiler, achieve an average efficiency of 0.69 for SP class 'A', 0.53 for SP class 'B', 0.87 for BT class 'A' and 0.99 for BT class 'B'. Despite good average efficiency for BT, Figure 2 shows that the PGI version has considerably lower scalability than the dHPF and MPI versions. The full array transposes cause communication volume proportional to the full *volume* of the partitioned multidimensional arrays, whereas communication volume for multipartitioning is proportional to the *surface area* of the partitioned tiles.

Previously, Frumkin et al. [17, 18] studied several parallel implementations of the NAS benchmarks on an SGI Origin 2000. As part of their study, they developed their own HPF version of the SP and BT benchmark codes for use with PGI's *pghpf* compiler. Like PGI's implementation of the SP and BT codes, the HPF versions of SP and BT implemented by Frumkin et al. used transposes between line sweeps to avoid communication within the sweeps. Frumkin et al. found their *pghpf*-compiled code to be less efficient and less scalable than the NAS hand-coded MPI parallelizations. Their findings were consistent with our own experiments, although our experiments with PGI's HPF codes showed them to be more efficient for smaller processor configurations than the results reported by Frumkin et al. for their HPF codes. Frumkin et al. [17] found the execution time of their HPF codes to be slower than the MPI hand-coded parallelizations by 62% for BT (class A) and by 150% for SP (class A) on 25 processors. In our experiments, we found that PGI's code was 30% slower for BT (class A), and 98% slower for SP (class A) on 25 processors. Both of these sets of experiments found that transposed-based parallelizations of SP and BT were not as efficient as parallelizations using multipartitioning. As a consequence, *pghpf*-generated code could not approach hand-coded performance.

Frumkin et al. [18] also compare the results of parallelizing the SP and BT codes with the CAPTools interactive parallelization tool [19] and SGI's proprietary shared-memory parallelizing compiler. Their findings show that, with user guidance, the CAPTools approach produces code yielding better performance than the corresponding *pghpf* compiled versions. Execution time for BT approaches that of the hand-coded MPI version, while for SP execution time lags by only 30%.

Serial versions of SP and BT annotated with SGI's shared memory parallelization directives and parallelized with SGI's compiler, also performed relatively well, but the generated code is suitable only for a tightly-coupled shared memory multiprocessor.

Compared to other compiler and tool-based parallelizations, dHPF produced code that yielded closest to the performance of the hand-coded versions of SP and BT. In the following sections, we analyze the contribution of key data-parallel compilation techniques and optimizations that dHPF applies to achieve this level of performance.

4. Data Partitioning Support

The previous section showed considerable variations in performance between codes using different partitioning strategies. Here, we quantitatively evaluate the impact of the partitioning strategy on overall execution time and communication volume. The data partitioning strategy employed in a parallel code influences an application's communication pattern, communication schedule, communication volume and local node performance. The data partitioning choice is a key determinant of an application's overall performance.

Here, we use the SP and BT benchmarks to compare the impact of three partitioning choices suitable for tightly-coupled line sweep applications. Tables 1 and 2 present ratios that compare the execution time and communication volume of the 2D-block partitioned versions of the codes and PGI's 1D-block partitioned versions of the codes with respect to dHPF's multipartitioned versions. Ratios greater than one indicate higher costs for the partitioning alternatives compared to multipartitioning.

Benchmark	16 proc.	64 proc.
Time SP 'A' CGP	1.45	1.23
Comm. Vol. SP 'A' CGP	1.09	1.08
Time SP 'B' CGP	1.76	1.58
Comm. Vol. SP 'B' CGP	1.09	1.12

Table 1: 2D-block CGP vs. multipartitioning.

Benchmark	16 proc.	64 proc.
Time SP 'A' PGI	1.40	1.61
Comm. Vol. SP 'A' PGI	3.29	3.27
Time SP 'B' PGI	1.91	1.98
Comm. Vol. SP 'B' PGI	4.28	3.28
Time BT 'A' PGI	0.85	1.34
Comm. Vol. BT 'A' PGI	3.11	3.16
Time BT 'B' PGI	0.95	1.69
Comm. Vol. BT 'B' PGI	4.00	3.13

Table 2: 1D-block transpose (PGI) vs. multipartitioning.

From Table 1, we see that the relative increase in communication volume with respect to multipartitioning is modest for both problem sizes. The increase in execution time is not directly

proportional to the increase in communication volume. Previous studies have shown that the principal factor driving the increase in execution time is that the coarse-grain pipelining version incurs more serialization than multipartitioning [14].

From Table 2, it is clear that the communication volume of PGI's 1D-block transpose implementation is a factor of 3–4 larger than that of dHPF's multipartitioned version. For smaller numbers of processors, this strategy is reasonably competitive, but for larger numbers of processors, efficiency degrades. Without more detailed analysis of PGI's generated code, the precise reasons for differences in scalability and performance in response to data and processor scaling are not clear.

5. Communication Optimizations

On modern architectures, achieving high-performance with a parallel application is only possible if processors communicate infrequently. The high cost of synchronizing and exchanging data drives the need to reduce the number of communications. This is true on NUMA-style distributed shared memory machines, multicomputers and clusters.

We have designed and implemented a broad range of communication analysis and optimization techniques in the dHPF compiler. These techniques improve the precision of processor communication analysis, reduce the frequency of messages between processors and reduce the volume of data communicated. We also describe some language extensions that assist the compiler in generating high performance code.

5.1 HPF/JA-inspired Extensions

The Japanese Association for High Performance Fortran proposed several new HPF directives to enable users to fine-tune performance by precisely controlling communication placement and providing a limited means for partially replicating computation. These directives are known as the HPF/JA extensions [20]. dHPF implements variants of the HPF/JA extensions for several purposes, as we describe below.

Eliminating Communication. To enable an HPF programmer to avoid unnecessary communication that can't be eliminated automatically by an HPF compiler without sophisticated analysis, dHPF supports the HPF/JA `LOCAL` directive. The `LOCAL` directive asserts to the compiler that communication for a set of distributed arrays (specified as parameters to `LOCAL`) is not needed in a particular scope.

Partially Replicating Computation. Enabling an HPF programmer to partially replicate computation to reduce communication can be important for high performance. For this reason, we provide extended support in dHPF for the `ON HOME` directive to enable users to partially replicate computation in shadow regions. This support was inspired by the HPF/JA `ON EXT_HOME` directive, but enables more precise replication control. The HPF/JA `ON EXT_HOME` directive causes computation to be partially-replicated for *all* elements in an array's shadow regions. A drawback of the `ON EXT_HOME` is that it can cause computation for elements in an arrays shadow region that are not needed. To avoid this undesirable effect, we extended HPF's `ON HOME` directive to allow multiple `ON HOME` references to be specified; this provides more precise control over partial replication than `ON EXT_HOME` and enables the manual specification of *partially replicated* computation partitionings similar to ones generated semi-automatically by the dHPF compiler for localization [4]. Figure 3 shows how using our extended `ON HOME` directive enables an application developer to selectively

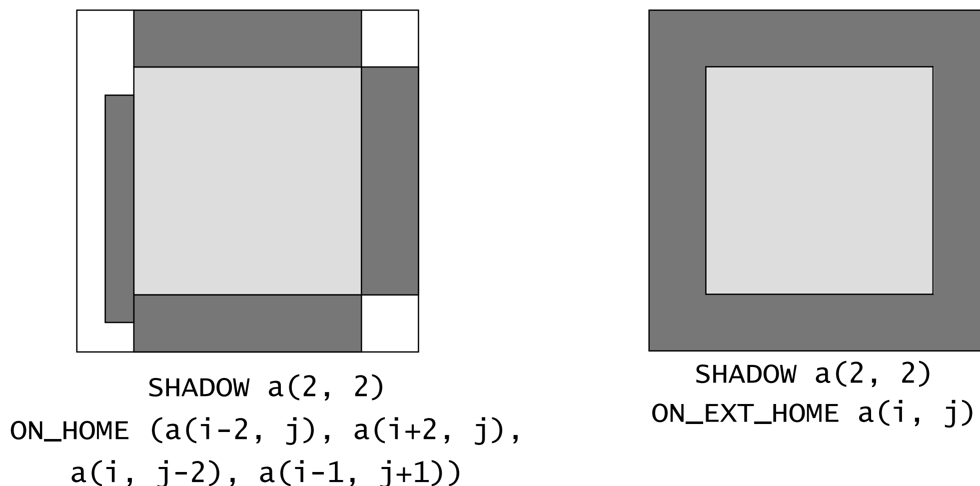


Figure 3: Extended `ON HOME` vs. `EXT_HOME`

partially replicate a computation to fill a portion of the shadow region (not all of the shadow region may be needed in a particular step of the computation). In the figure, each light gray square represents an owned section for a processor; the enclosing squares represent the processor's shadow region. dHPF's extended `ON HOME` directive enables us to include or exclude the "corners" that would be filled by the HPF/JA `EXT_HOME` directive. Dark gray sections represent areas where computation was partially replicated using the directives shown in the figure.

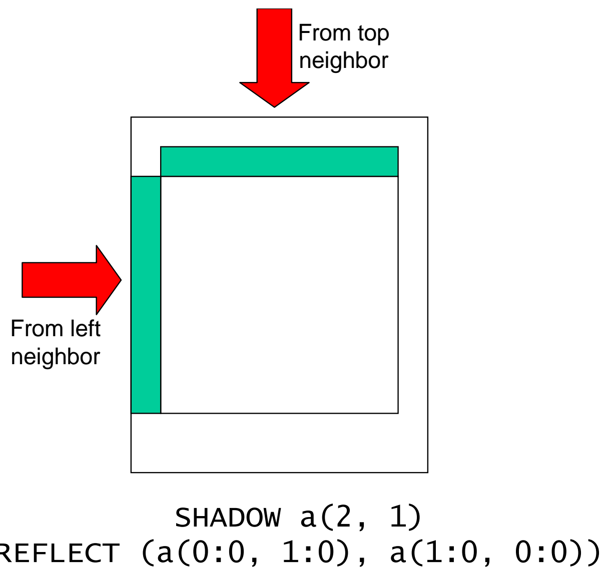
Enabling Explicit Communication. The HPF/JA `REFLECT` directive was designed to support explicit data replication into shadow regions of neighbors. `REFLECT` was designed for use in conjunction with the `LOCAL` directive to avoid redundant communication of values. At the point a `REFLECT` directive occurs in the code, communication will be scheduled to fill each processor's shadow regions for a distributed array with the latest values from neighboring processors. In dHPF, we implemented support for an extension of the `REFLECT` directive that enables more precise control over the filling of the shadow regions. Our extension to `REFLECT` enables the programmer to specify the dimensions and the depth of an array's shadow regions to fill. By specifying a single dimension and width, we can *selectively* fill all or part of the shadow region along that dimension. If corner elements are needed, multiple dimensions can be specified at once in a single entry, which will include them. Figure 4 shows 2D array `a`'s left and top shadow regions selectively filled in from the values owned by its left and top neighbors.

In the following sections, we describe how we use these directives to tune application performance.

5.2 Partially Replicated Computation

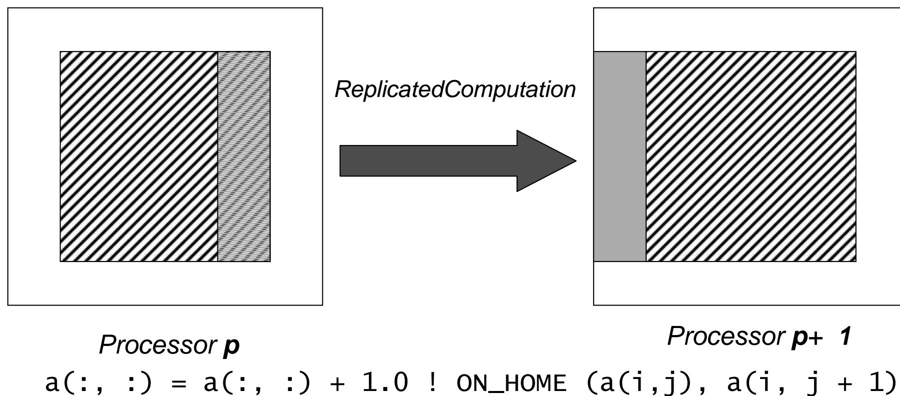
The dHPF compiler's ability to compute statements on multiple home locations enables it to partially replicate computation. This approach, used judiciously, can significantly reduce communication costs and even, in some cases, eliminate communication altogether for certain arrays.

With dHPF, computation can be partially replicated either automatically by the compiler [4], or manually using the extended `ON HOME` directive. Figure 5 illustrates the technique for a simple case: computation for a panel of columns has been replicated on neighboring processors (the owning




```
SHADOW a(2, 1)
REFLECT (a(0:0, 1:0), a(1:0, 0:0))
```

Figure 4: Extended `REFLECT` directive




```
a(:, :) = a(:, :) + 1.0 ! ON_HOME (a(i,j), a(i, j + 1))
```

Figure 5: Partially Replicated Computation along a processor boundary

processor and the non-owning one) by using an extended computation partitioning as described in section 5.1. The figure shows the locally-allocated sections of distributed array `a` for two different processors. The diagonally-striped regions represent the each processor's "owned" section; the squares enclosing each owned section represent surrounding shadow regions. The shaded portion of processor $p+1$'s shadow region represents the replicated computation of one column owned by processor p, as shown by the shaded portion of processor p's owned region.

For SP and BT, partial computation replication enables us to completely eliminate boundary communication for several large distributed arrays, without introducing any additional communication. In the BT benchmark, partial computation replication eliminates communication for 5D arrays, `fjac` and `njac` in the `lhsx`, `lhsy`, and `lhsz` routines. In both the BT and SP benchmarks, partial computation replication completely eliminates communication for six 3D arrays in the `compute_rhs` routine. The computation of these values can be replicated at boundaries using

values of a 4D array that must be communicated anyway. The hand-coded parallelizations exploit this partial replication of computation too.

5.2.1 Performance Impact

Table 3 compares the relative performance of versions with and without partial replication of computation. We present the ratios of execution time and communication volume between the dHPF-generated multipartitioned versions of SP and BT without partial computation replication and those with partial replication. Ratios greater than one indicate the cost of not using partial replication. The partially replicated versions use both compiler-derived and explicitly-specified replication using the extended `ON HOME` directive. The results in Table 3 show that without partial replication of computation, both execution time and communication volume increase significantly.

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.18	1.36
Comm. Vol. SP 'A'	1.33	1.36
Time SP 'B'	1.12	1.21
Comm. Vol. SP 'B'	1.33	1.35
Time BT 'A'	1.35	1.58
Comm. Vol. BT 'A'	2.94	2.96
Time BT 'B'	1.13	1.62
Comm. Vol. BT 'B'	2.97	2.97

Table 3: Impact of partial computation replication

This optimization contributes significantly to scalability: its relative improvement in execution time is more significant when communication time is not dominated by computation; for this reason, we observe a higher impact for the smaller class 'A' problem size on a large number of processors. It is worth noting that our experiments were executed on a tightly-coupled parallel computer with a high-bandwidth low-latency interconnect; this reduces the performance penalty associated with higher message volume.

5.2.2 Interprocedural Communication Elimination

dHPF's communication analysis and generation is procedure-based; dHPF does not currently support interprocedural communication analysis and placement. However, using our HPF/JA-inspired directive extensions enables us to eliminate communication of values that we know are available as a side effect of partially-replicated computation or communication elsewhere in the program.

We present results on the impact of using the HPF/JA directives to eliminate communication across procedures on the selected benchmarks. Table 4 presents ratios comparing the execution time and communication volume of dHPF-generated multipartitioned versions of SP and BT without using the HPF/JA directives to eliminate communication across procedures with respect to the versions that use them. Ratios greater than one indicate increases in execution time and communication volume. Without the directives, communication volume and execution time are roughly 10–20% higher.

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.05	1.15
Comm. Vol. SP 'A'	1.12	1.13
Time SP 'B'	1.05	1.08
Comm. Vol. SP 'B'	1.11	1.12
Time BT 'A'	1.12	1.09
Comm. Vol. BT 'A'	1.18	1.18
Time BT 'B'	1.06	1.20
Comm. Vol. BT 'B'	1.18	1.18

Table 4: Impact of using the HPF/JA directives to eliminate communication interprocedurally.

In SP, we use the `LOCAL` directive to eliminate communication for three 3D arrays in the `lhsx`, `lhsy` and `lhsz` routines for values that are available locally because their computation was partially replicated in the `compute_rhs` routine using extended `ON HOME`.

In BT, we use the `LOCAL` directive to eliminate communication for a 3D array `u` in the `lhsx`, `lhsy` and `lhsz` routines. The required off-processor values for `u` had already been communicated into the shadow region in routine `compute_rhs` by a `REFLECT` directive.

The combination of the HPF/JA `LOCAL`, extended `ON HOME` and `REFLECT` directives obtain all of the benefits of interprocedural communication analysis, without its complexity.

5.3 Coalescing

For best performance, an HPF compiler should minimize the number and volume of messages. Analyzing and generating messages for each non-local reference to a distributed array in isolation produces too many messages and the same values might be transferred multiple times.

In dHPF, a communication set is initially computed and placed separately for each individual non-local reference. A communication set is represented in terms of an `ON HOME` reference (corresponding the computation partitioning where the data is required) and a non-local reference. Both references are represented in terms of value numbers that appear in their subscripts (if any). Whenever possible, dHPF vectorizes communication and hoists it out of loops. When multiple communication events are scheduled at the same location in the code, dHPF tries to coalesce them to avoid communicating the same data multiple times.

Consider the code in Figure 6, an HPF compiler should generate a single message to communicate a single copy of the off-processor data required by both references to `b(i, j - 1)`. However, detecting when sets of non-local data for multiple references overlap is not always so simple. References that are *not* syntactically equivalent may require identical or overlapping non-local data. In Figure 7, references `b(i, j - 1)` and `b(i, j)` require identical non-local values and can be satisfied by a single message. To avoid communicating duplicate values, overlap between sets of non-local data required by different loop nests should be considered as well, as shown in Figure 8.

5.3.1 Normalization

To avoid communicating duplicate values, dHPF uses a normalization scheme as a basis for determining when communication sets for different references overlap. To compensate for differences

```
CHPF$ distribute a(*, block), b(*, block) onto P
do j = 2, n
  do i = 1, n
    a(i, j) = b(i, j - 1) + c ! ON_HOME a(i, j)
    a(i, j) = a(i, j) + d + b(i, j - 1) ! ON_HOME a(i, j)
  end do
end do
```

Figure 6: Simple overlapping non-local data references.

```
CHPF$ distribute a(*, block), b(*, block) onto P
do j = 2, n
  do i = 1, n - 1, 2
    a(i, j) = b(i, j - 1) + c ! ON_HOME a(i, j)
    a(i + 1, j) = d + b(i, j) ! ON_HOME a(i + 1, j)
  end do
end do
```

Figure 7: Complex overlapping non-local data references

in computation partitionings selected for different statements, normalization rewrites the value numbers representing a communication set into a form relative to its ON_HOME reference expressed in a *canonical* form. In our discussion of normalization, we refer to the ON_HOME reference for a communication set as the *computation partitioning (CP) reference* and the non-local reference as the *data reference*.

dHPF's value-number based representation for communication sets has the disadvantage that non-local references that arise in different loops are incomparable because their subscripts have different value number representations. To enable us to detect when such references may require overlapping sets of non-local data, we convert all data and CP references to use a canonical set of value numbers for the loop induction variables involved.

We apply our normalization algorithm to communication sets represented in terms of value numbers based on affine subscript expressions of the form $ai + b$, where i is an induction variable, a is a known integer constant and b is a (possibly symbolic) constant. This restriction comes from the need to compute a symbolic inverse function for such expressions[2]. If this restriction is not met, the communication set is left in its original form.

A communication set is in normal form if:

- The CP reference is of the form $A(i_1, i_2, i_3, ..., i_n)$
- The data reference is of the form $A(a_1 i_1' + b_1, a_2 i_2' + b_2, a_3 i_3' + b_3, ..., a_n i_n' + b_n)$

where A is an n-dimensional array, each i_k is an induction variable or a constant, and each i_j' corresponds to a unique i_k. We say that a particular subscript expression in the CP reference is normalized if it is of the form i_j, where i_j is an induction variable or a constant.

If a communication set is not in normal form but meets our restriction of affine subscript expressions, we normalize it by computing symbolic inverse functions for each non-normalized CP

2. Normalization could be extended to support more complex affine expressions without too much difficulty.

```
do timestep = 1, T
          Coalescedataexchangeatthispoint
  do j = 1, n
    do i = 3, n
      a(i, j) = a(i + 1, j) + b(i - 1, j) ! ON_HOME a(i,j)
    enddo
  enddo

  do j = 1, n
    do i = 1, n - 2
      a(i + 2, j) = a(i + 3, j) + b(i + 1, j) ! ON_HOME a(i + 2, j)
    enddo
  enddo

  do j = 1, n
    do i = 1, n - 1
      a(i + 1, j) = a(i + 2, j) + b(i + 1, j) ! ON_HOME a(i + 1, j)
    enddo
  enddo
enddo
```

Figure 8: Coalescing non-local data across loops.

subscript position. We then apply this function to each subscript position in both the CP and data references. After this step, *only* the data reference has subscripts of the form $a_j i_j + b_j$. Each subscript position gets a canonical value number represented by an artificial induction variable that has the range and stride of the original $a_j i_j + b_j$ subscript expression. We apply this normalization step to all communication events at a particular location in the code before attempting to coalesce them.

5.3.2 Coalescing Normalized Communication Sets

Following normalization, we coalesce communication events to eliminate redundant data transfers. There are two types of coalescing operations that can be applied to a pair of communication events:

- Subsumption: Identify and eliminate a communication event (nearly) completely covered by another.

- Union: Identify and fuse partially overlapping communication events, which do not cover one another.

Both operations eliminate communication redundancy, however these two cases are handled separately at compile time.

For both cases, the coalescing algorithm first tests to see if communication events are compatible. To be compatible, both communication events must correspond to the same distributed array, be reached by the same HPF alignment and distribution directives, have the same communication type (read or write) and be placed at the same location in the intermediate code.

Subsumption The subsumption of one communication event by another requires that the non-local data of the subsuming event be a superset of that of the subsumed one. According to the model for normalized data and CP references, this implies that both messages represent data shifts of constant width in one or more dimensions. For example, a single-dimensional shift of a submatrix owned by a processor would correspond to sending a few rows or columns to a neighboring processor. A pair of shifts must be in the same direction and along the same dimensions for communication events to be compatible. For instance, trying to coalesce a shift that sends two rows and a shift that sends two columns of a submatrix is infeasible.

For two compatible shift communications, where the *shift width* of one is larger than that of the other, the volume of data transferred can be reduced by completely eliminating the smaller shift since its data values will be provided by the larger one. For dimensions not involved in a shift, the ranges of the data reference subscript value numbers in the subsuming communication event must be supersets of the corresponding ranges in the subsumed event. For data dimensions involved in a shift, subsumption does not require that their range be a strict superset of the corresponding ranges in the subsumed event. If the subsuming communication has a wider shift width, but doesn't have a large enough range for its loop induction variable, the coalescer synthesizes a new induction variable with extended range to cover the induction variable in the subsumed event as well. This range-extension technique is very effective for avoiding partially-redundant messages and generating simple communication code.

Union Coalescing partially overlapping communication events requires less strict conditions than subsumption. Given normalized data and CP references, The coalescer will only try to union communication events that have common shift dimensions and directions.

The dHPF compiler uses the Omega library [21] to implement operations on integer tuples for its communication analysis and code generation [5]. We apply integer set-based analysis to determine the profitability of unioning two communication events. We construct sets that represent the data accessed by the non-local references of each communication event. If these sets intersect, this implies that the communication sets for some pair of processors *may* intersect. In this case, the algorithm will coalesce the two communication events by unioning them. If the sets do not intersect, then the communication sets for any pair of processors also do not intersect, which implies there is no redundancy to eliminate, so coalescing is not performed.

As described in the previous section, generation of a coalesced communication set can require that new induction variables be synthesized with extended range. When unioning communication sets, this transformation is applied to an induction variable appearing in any dimension as necessary.

5.3.3 Performance Impact

Table 5 presents the relative performance of codes where coalescing was applied with and without normalization. (Coalescing without normalization may overlook opportunities for eliminating redundancy.) We present the ratios of execution time and communication volume between the dHPF-generated multipartitioned versions of SP and BT with un-normalized coalescing and those with normalized coalescing. Ratios greater than one indicate the overhead of un-normalized coalescing compared to normalized coalescing. The results show that without normalization, both execution time and MPI communication volume increase.

We found that the reduction in execution time due to normalized coalescing is mostly due to reduced communication volume in the critical tightly-coupled line sweep routines (`x_solve`,

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.41	2.10
Comm. Vol. SP 'A'	1.70	1.71
Time SP 'B'	1.28	1.65
Comm. Vol. SP 'B'	1.71	1.71
Time BT 'A'	1.02	1.24
Comm. Vol. BT 'A'	1.32	1.32
Time BT 'B'	1.04	1.08
Comm. Vol. BT 'B'	1.32	1.33

Table 5: Impact of normalized coalescing.

`y_solve`, `z_solve`) of SP and BT. Loosely coupled communication is also reduced, but its impact is not as significant on machines with a high-bandwidth, low-latency interconnect such as the Origin 2000. The greater execution time of the un-normalized version for SP cannot be attributed completely to the extra communication volume, because un-normalized coalescing produces sets which are not rectangular sections; this leads to less efficient management of communicated data, as explained in more detail in Section 6.2.

After both partial replication of computation and normalized coalescing, we found that the communication volume for the dHPF-generated code for SP was within 1% of the volume for the hand-coded versions. For BT, we found that the volume in the dHPF-generated code was actually lower by 5% (class B) and 17% (class A) than that of the hand-coded versions.

5.4 Multi-variable Aggregation

Often, applications access multiple arrays in a similar fashion. When off-processor data is needed, this can lead a pair of processors to communicate multiple variables at the same point in the program. Rather than sending each variable in a separate message, dHPF reduces communication overhead by shipping multiple arrays in a single message in such cases. dHPF's implementation strategy of aggregation is described elsewhere [14]. Here we focus on message aggregation's impact on performance. Table 6 presents ratios for execution time and communication frequency that compare the cost of of non-aggregated versions of the codes with respect to aggregated versions. We present the ratios of execution time and communication frequency between the dHPF-generated multipartitioned versions of SP and BT without aggregation and those with aggregation. Ratios larger than one indicate an increase in cost without aggregation. While aggregation reduces message frequency considerably for both SP and BT in all cases, the impact of aggregation on execution time for BT is small because of its high computation to communication ratio. However, the latency reduction that aggregation yields eventually becomes important when scaling a computation to a large enough number of processors so that communication time becomes significant with respect to computation. SP has a higher communication/computation ratio and thus aggregation has a larger impact. For SP, as we drop back from the larger class 'B' to the smaller class 'A' problem size, and/or increase the number of processors, the execution time impact of aggregation increases. For the 64 processor execution of SP using the class 'A' problem size, aggregation cuts execution time by 10%.

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.02	1.10
Comm. Freq. SP 'A'	2.12	2.31
Time SP 'B'	1.01	1.03
Comm. Freq. SP 'B'	2.12	2.31
Time BT 'A'	1.00	1.01
Comm. Freq. BT 'A'	1.37	1.43
Time BT 'B'	1.03	1.02
Comm. Freq. BT 'B'	1.37	1.43

Table 6: Impact of aggregation.

6. Memory Hierarchy Issues

Today's computer systems rely on multi-level memory hierarchies to bridge the gap between the speed of processors and memory. To achieve good performance, applications must use the memory hierarchy effectively.

In the dHPF compiler, we have implemented many techniques to improve memory hierarchy utilization. These include padding of dynamically allocated arrays, inter-array tile grouping (which lays out corresponding tiles from different arrays consecutively in memory to reduce conflicts), and an arena-based communication buffer management scheme to reduce the cache footprint of communication buffers. Here we discuss an optimization that simplifies generated code for enumerating communication sets, a novel compiler-based technique for managing off-processor data that increases cache efficiency and the tile grouping strategy. We also present an analysis of the negative impact on performance of our array padding technique.

6.1 Processor Set Constraints

The dHPF compiler can generate code for a symbolic processor count. To do so, it must analyze and generate communication for data distributions partitioned into blocks whose sizes are symbolic [5]. There are significant challenges that must be met in order to generate fast and efficient code for this case.

As described in [5], the principal challenge for generating efficient code comes from the presence of a symbolic block size and its implications inside our integer-set based compiler core. A precise mapping of symbolically-sized blocks to processors is not possible in the Presburger arithmetic formalism underlying the Omega library; such a mapping can only be described with products of two symbolic variables: the processor id and the symbolic block size.

For example, distributing a 1D array of extent N over P processor would yield a block size of $B = \lceil N/P \rceil$. Figure 9 illustrates a processor grid with these characteristics. The mapping that naturally represents the set of template elements owned by processor p can be expressed as:

$$\{[t] \rightarrow [p] : Bp + 1 \leq t \leq Bp + B \wedge 1 \leq t \leq N \wedge 1 \leq p \leq P\} \tag{1}$$

where t represents a template element, B is the block size and p is a particular processor. The Bp product terms are *not* legal in Presburger arithmetic and thus this formula cannot be represented directly using Omega's integer tuples.

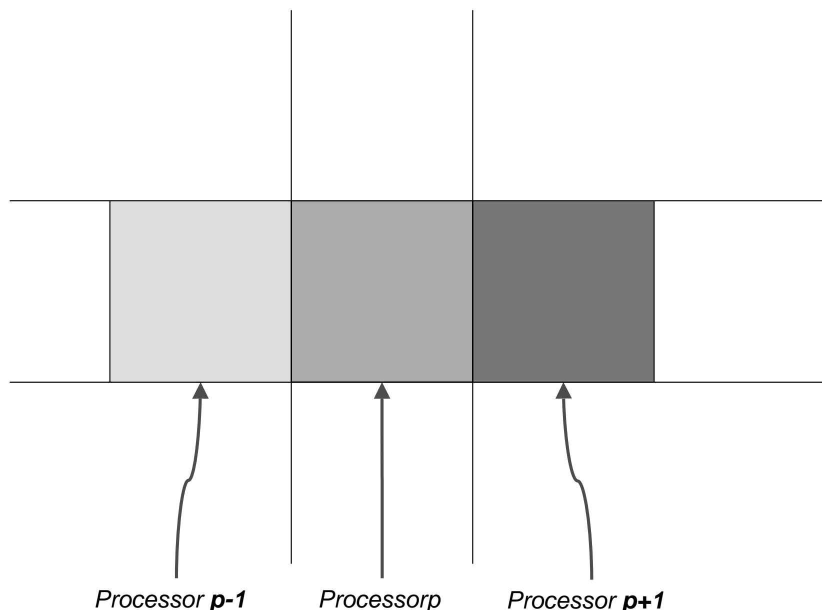


Figure 9: Processor Grid

For this reason, dHPF represents such data distributions less precisely using a virtual processor formulation that avoids symbolic multiplication. The set of elements owned by a virtual processor v can be expressed as:

$$\{[t] \rightarrow [v] : v \leq t \leq v + B - 1 \wedge 1 \leq t \leq N \wedge 1 \leq v \leq N\} \quad (2)$$

where t represents a template element, B is the block size and v represents a virtual processor. This formula represents the mapping of data to a grid of virtual processors, with a virtual processor aligned on every template element. However, this formulation fails to precisely represent the relationship between the data tiles owned by the real processors. To generate correct code for enumerating processor partners with this representation, dHPF first uses the Omega library to generate loops that enumerate every possible virtual partner, then the compiler transforms the loops to enumerate only real processor partners. For complex communication sets, this process leads to complex code. To reduce the complexity of the generated code, we augmented dHPF to introduce constraints that sharpen information about relationships between processors, outside of the virtual processor mapping formulation.

6.1.1 Necessity of Constraints

The virtual processor data mappings described previously, are used to derive the partner processors with which a particular processor p has to exchange data. These sets are overly general because they represent possible communication with any partner on the virtual processor grid, *not* with the real partners that will exist at runtime.

Figure 10 shows a two-dimensional BLOCK style data distribution and shows possible communication with a *right* neighbor of processor p. If the compiler can determine that the communication event is a single-dimensional shift, then there is no need to contemplate possible virtual proces-

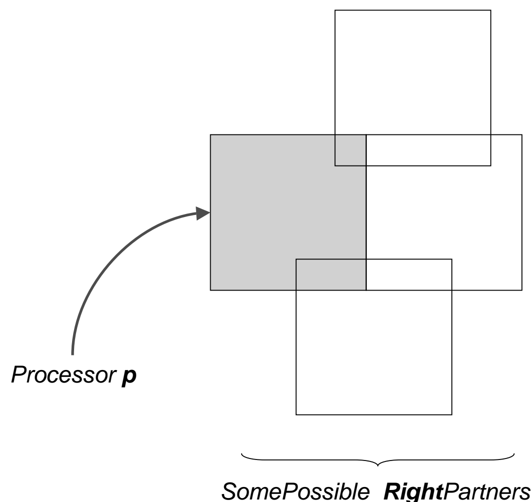


Figure 10: Unconstrained Right Partners

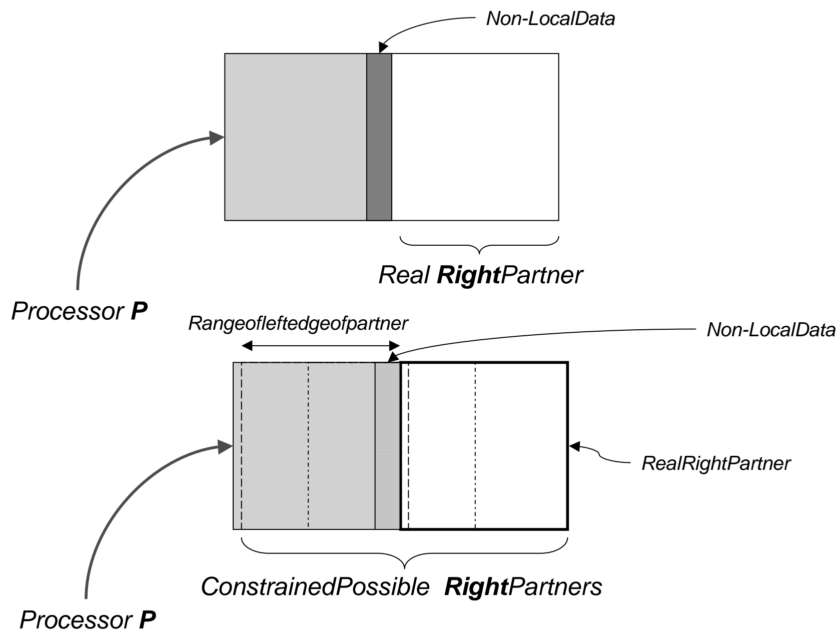


Figure 11: Constrained Right Partners

sors that have a coordinate that differs on the *other* dimension. The compiler can assert that the coordinate in the *other* dimension on the partner processor are the same as those on p.

More precisely, if processor p has coordinates $(p_0, p_1, ..., p_{d-1})$ in a d-dimensional processor grid and the communication event has been determined to occur only along dimension i, then if the partner processor is w with coordinates $(w_0, w_1, ..., w_{d-1})$, we can add the following constraints to the integer set that represents the potential virtual partner processors:

$$\bigwedge_{\forall j \neq i} p_j = w_j \tag{3}$$

These constraints do not require the introduction of symbolic products, and thus can be used in conjunction with dHPF's virtual processor representation to simplify the analysis of and code generation for processor sets.

Figure 11 represents a constrained virtual processor set for a *right* shift communication. The set still contains imprecision in the horizontal dimension, but we have eliminated the imprecision in the vertical dimension.

Adding processor set constraints is also necessary to make integer-set based analysis and code generation techniques efficient at compile time.

6.1.2 Performance Impact

Table 7 presents results showing the impact of processor constraints on the SP and BT benchmarks. The table shows the relative performance of versions of the codes without processor set constraints, compared to the constrained versions. Ratios greater that one indicate the runtime overhead of not using processor constraints. The metrics presented are execution time, mispredicted branches and L2 instruction cache misses. These metrics were chosen because additional cache misses and mispredicted branches show the runtime effects of code complexity. Without processor set constraints, the generated code is considerably more complex with many more conditionals and loops.

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.01	1.04
Mis. Branches SP 'A'	0.99	1.01
L2 I-Cache Miss. SP 'A'	1.16	1.48
Time SP 'B'	1.03	1.07
Mis. Branches SP 'B'	0.92	0.99
L2 I-Cache Miss. SP 'A'	1.05	1.13
Time BT 'A'	0.98	1.02
Mis. Branches BT 'A'	1.12	1.10
L2 I-Cache Miss. SP 'A'	1.09	1.21
Time BT 'B'	1.00	1.00
Mis. Branches BT 'B'	0.93	0.99
L2 I-Cache Miss. SP 'A'	1.04	1.02

Table 7: Impact of processor set constraints.

Adding processor set constraints has a small impact in execution time; its main impact is reducing the size of the code and the number of processor loops that have to be evaluated. This reduces the number of instruction cache misses that have to be taken as well as the number of mispredicted branches (in some cases). The main effect of this optimization is that it speeds up compilation, since adding the processor constraints converts the processing of multidimensional communication sets into processing of single dimensional sets [14, 22].

6.2 Overlap Regions vs. Direct-Access Buffers

Overlap regions [23], in which a processor allocates additional storage around the boundary of data it owns to store neighboring off-processor data, are commonly used by compilers and application developers. The dHPF compiler allows the use of overlap regions for distributed arrays. An HPF2

SHADOW directive specifies the extent of an overlap region for an array on a dimension-by-dimension basis. Figure 12 shows HPF source code with an overlap region specified for array a; the SHADOW directive specifies a lower overlap of width one in the second dimension.

```
CHPF$ distribute a(*, block) onto P
CHPF$ shadow a(0, 1:0)
do j = 2, n
  do i = 1, n
    a(i, j) = a(i, j - 1) + c ! ON_HOME a(i, j)
  end do
end do
```

Figure 12: An overlap region on an array in HPF.

Overlap regions are convenient because they enable uniform access to both local and off-processor data, which leads to simple code for partitioned loops. For example, in Figure 13 overlap regions make it possible to access a(i, j - 1) for j equal to the processor boundary (my_j_lo).

```
do j = max(2, my_j_lo), min(my_j_lo + j_blocksize, n)
  do i = 1, n
    a(i, j) = a(i, j - 1) + c
  end do
end do
```

Figure 13: SPMD code using an overlap region.

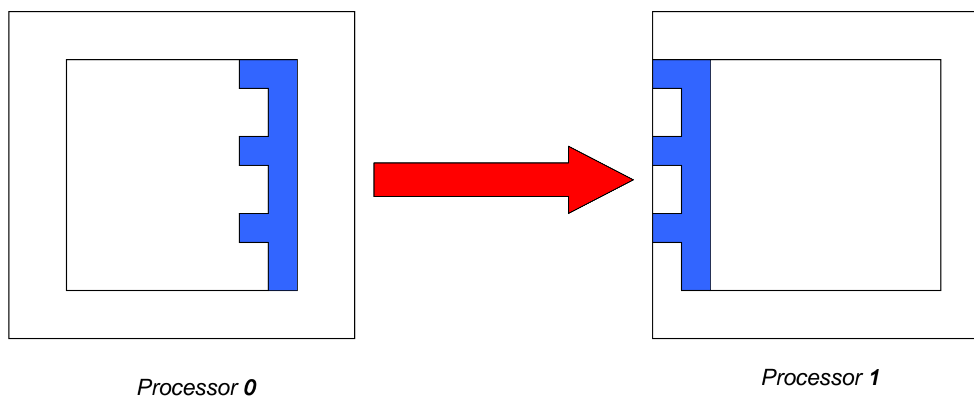


Figure 14: Overlap Regions

Figure 14 shows a 2D overlap region around a processor's data tile, in this case processor 0 is shifting a strided rectangular section of data to processor 1. Communication of this data section needs several copies: first the data section needs to be copied into a buffer (it may have been aggregated with compatible sections of other arrays), then the receiver (processor 1) has to unpack the data from the buffer into the overlap region, so the computation can access it from there.

The diagram shows an example with just a *right* horizontal shift phase, but an application might have a *left* horizontal shift phase and *bottom* and *up* vertical shift phases, which use the other overlap

```
do j = max(2, my_j_lo), min(my_j_lo + j_blocksize, n)
  do i = 1, n
    if (j - 1 .lt. my_j_lo) then ! REMOTE data
      t1 = buffer_a(i, j - my_j_lo)
    else ! LOCAL data
      t1 = a(i, j - 1)
    end if
    a(i, j) = t1 + c
  end do
end do
```

Figure 15: SPMD code using a direct-access buffer.

regions in a similar manner. If we assume a column-major layout, then for each *separate* phase, the elements in the sections of the overlap region which are not used create unnecessary holes in the cache.

While overlap regions lead to simple SPMD code, there are three ways that using them can degrade performance. First, any loop accessing an array that has overlap regions allocated, which does not use most of the overlap region in each of the dimensions for which it is provided, suffers from both reduced spatial reuse (values in the overlap region may be fetched into cache and not used) and less effective cache utilization (some cache sets may be underutilized because unaccessed overlap regions map to them). Second, if data is received into a message buffer and then copied into overlap regions, there will be two live copies of the data occupying space in the cache. Third, copying the data from a communication buffer into an overlap region can be costly, particularly if the data in the overlap region is non-contiguous.[3]

To avoid the cache inefficiency that comes with using multi-dimensional overlap regions for single-dimensional shift communication, the dHPF compiler supports accessing remote data directly out of communication buffers. This has the dual advantages of eliminating the unpacking phase on the receiving processor as well as eliminating the need for overlap regions on the receiving processor's tile. Directly accessing data out of communication buffers, introduces two modes of access for array references: boundary remote data, accessed out of the buffer and interior data accessed out of the array.

If the compiler were to generate naive code for data that may reside either in a buffer or in a local array, each access would require a conditional test, which could be costly. Figure 15 shows naive code for this situation. To avoid the overhead of this approach, the dHPF compiler splits a loop nest that accesses non-local data out of buffers into a loop nest whose iterations *may* require remote data and those that *must* access data only from the local array.

Loop splitting in this manner eliminates conditionals from the interior section of the loop nest, but may not eliminate conditionals in the non-local section of the loop nest. Figure 16 shows a split loop with one iteration space accessing data for array `a` only out of the local array section, and the other iteration space accessing data out of a buffer. In this case, it was possible to eliminate all conditionals for the non-local reference, because the set of non-local iterations for the statement is a subset of the non-local iterations for the reference. In a more general case, the non-local iteration space may not access exclusively off-processor data; a conditional would be required in this case.

3. *Any* data movement in modern machines is costly!

```
do j = my_j_lo, my_j_lo + 1
  do i = 1, n
    a(i, j) = buffer_a(i, j - my_j_lo) + c
  end do
end do

do j = my_j_lo + 2, min(my_j_lo + j_blocksize, n)
  do i = 1, n
    a(i, j) = a(i - 1, j) + c
  end do
end do
```

Figure 16: Optimized use of a direct-access buffer.

6.2.1 Aggregation and Direct Buffer Access

Direct buffer-access is very useful, especially in combination with communication aggregation. Incoming non-local data for different arrays or disjoint sections of the same array is laid out sequentially in the buffer. dHPF generates code for directly accessing such data by using pointers into each contiguous section in the buffer. With this scheme, the dHPF compiler supports direct access to buffers comprised of *unions of constantly-strided rectangular sections*.

6.2.2 Performance Impact

We compare the efficiency of using direct-access buffers and overlap regions for the selected benchmarks. Table 8 presents ratios that compare the execution time and L2 cache misses for versions using overlap regions with respect to versions using direct access buffers. We present the ratios of execution time and L2 data cache misses between the dHPF-generated multipartitioned versions of SP and BT without direct buffer access and those with it. Ratios greater than one indicate the overhead of using overlap regions instead of buffers. Both the execution time and L2 cache misses increase without direct access buffers.

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.13	1.40
L2 Misses SP 'A'	1.22	1.41
Time SP 'B'	1.19	1.10
L2 Misses SP 'B'	1.09	1.11
Time BT 'A'	1.02	1.03
L2 Misses BT 'A'	1.01	1.09
Time BT 'B'	1.04	1.04
L2 Misses BT 'B'	1.00	1.01

Table 8: Impact of direct-access buffers on SP and BT.

Direct-access buffers play a significant role in reducing L2 data cache misses by avoiding extra copies into shadow regions and reducing the memory footprint of large multidimensional arrays.

In particular, they are very important for the tightly-coupled line sweep phases of the SP and BT benchmarks and the `lhs` and `rhs` arrays they use.

6.3 Tile-based Layout for Multipartitioned Arrays

Storing data for the local portion of any non-`BLOCK` partitioned array involves choosing a storage layout for multiple data tiles of the same array on each processor. This is the case for cyclically distributed arrays in which every k-th element (or every k-th group of n elements) is owned by the processor.

The same issue arises to a lesser degree with multipartitioned arrays: each processor owns a set of `BLOCK`-like tiles. Each individual tile can be laid out using standard Fortran column-major ordering, but tiles can be ordered in different ways. Figure 17 shows the set of data tiles owned by a particular processor for two different 2D multipartitioned arrays.

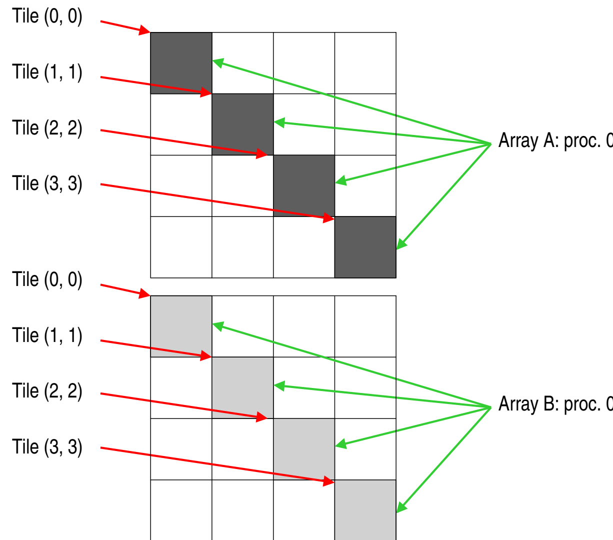


Figure 17: Multipartitioned Arrays

The obvious way to lay out multipartitioned data tiles in memory is to assign contiguous space for all of the tiles for each array. Figure 18 shows the tiles for the same two arrays laid out in memory: all tiles for each array are contiguous.

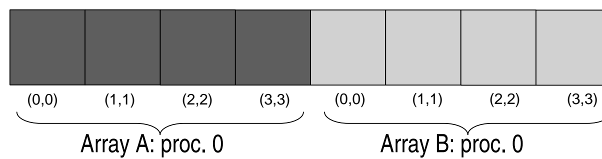


Figure 18: Contiguous tiles for separate arrays

The strategy of laying out tiles in the contiguous-by-array fashion has the potential for cache conflicts between simultaneously used tiles of different arrays. In line sweep computations, tiles of different arrays are used in the same order, i.e. the computation uses the *first* tile of both arrays, then the *second* tile of both arrays, etc. If the local portion of an array (sum of the sizes of all

tiles) is larger than the size of the cache then the possibility of conflicts between tiles of different arrays that are used together, increases.

A solution to this problem is to lay out in memory, the tiles that are used *together*. That is, lay out contiguously the *first* tile of each array, then the *second* tile of each array, etc. Figure 19 shows tiles for two different arrays, arranged in memory so that tiles from different arrays, which are commonly used together, are contiguous. This decreases the likelihood of cache conflicts between tiles belonging to different arrays, as well as increases spatial reuse. The same strategy can be applied for programs with more than two arrays.

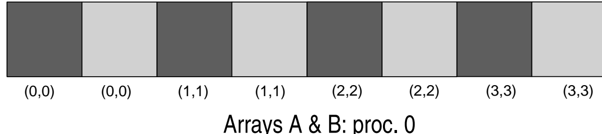


Figure 19: Contiguous simultaneously used tiles

6.3.1 Performance Impact

Table 9 shows the comparative performance of dHPF-generated multipartitioned versions of SP and BT using the regular layout with contiguous tiles for *each* array, relative to versions using the interleaved tile layout. We present the ratios of execution time and L2 data cache misses. Ratios greater than one indicate the cost of using the contiguous layout compared to the interleaved layout.

Benchmark	16 proc.	64 proc.
Time SP 'A'	0.99	1.00
L2 Misses SP 'A'	1.02	1.01
Time SP 'B'	1.02	1.00
L2 Misses SP 'B'	1.01	0.81
Time BT 'A'	1.04	1.32
L2 Misses BT 'A'	1.00	1.06
Time BT 'B'	1.11	1.07
L2 Misses BT 'B'	1.01	1.01

Table 9: Impact of tile-based for layout for SP and BT.

The table shows that the simultaneously-used tile based layout is a significant optimization reducing execution time by an average of 7% across all versions. This is due to a reduction of expensive L2 data cache misses. The only case in which it increases run time is for SP class A on 16 processors, but by only 1%. The number of L2 misses for SP class B on 64 processors, is anomalous because an *increase* in L2 misses (using the interleaved layout) did not increase execution time proportionally. The reason behind this anomaly requires more investigation of the behavior of other related metrics (L1 data misses, TLB misses) on this particular configuration.

6.4 Padding for Dynamically Allocated Arrays

Due to its support for symbolic processor counts, dHPF manages storage for each processor using dynamic memory allocation. Each processor allocates a contiguous region of memory for its local

portion of the distributed arrays. In particular, all multipartitioned arrays are allocated as a contiguous block of memory with a total size corresponding to the sum of the individual tile sizes of all arrays. Since dHPF performs source-to-source compilation, it uses Fortran's standard column-major data layout for the local portions of distributed arrays. Figure 20 shows an illustration of the data layout for two 2D data tiles on a processor. The tiles are allocated contiguously in memory, they *may* belong to the same distributed array or to different arrays, as explained in section 6.3.

To minimize intra-array cache conflicts, dHPF's runtime pads each distributed dimension of an array to an *odd* length. This reduces the possibility of L1 data cache conflicts between columns of the same tile (in general, hyperplanes belonging to the same tile). For tiles of a multipartitioned array, the same padding is applied to *every* tile. Padding plus overlap regions that are unused in a particular loop nest increase the number of unused data locations that can occupy address space also mapped to useful data, leading to less effective spatial and temporal reuse; we believed that a reduction in conflict misses from padding would offset this cost.

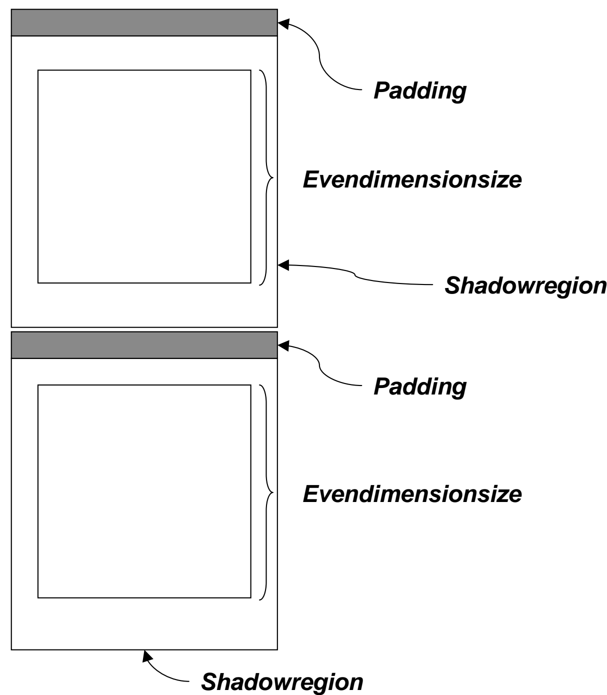


Figure 20: Array Padding

6.4.1 Performance Impact

Table 10 shows the comparative performance of dHPF-generated multipartitioned versions of SP and BT without padding, relative to versions *with* dynamically allocated array padding. We present the ratios of execution time, L1 data cache misses and L2 data cache misses. Ratios greater than one indicate the cost incurred when not using padding.

The results in table 10 show that padding only significantly decreases execution time for the SP class A benchmark on 16 processors. The improvement comes from a 165% decrease in L1 data cache misses using padding. Performance improves for SP class A because of the reduction

Benchmark	16 proc.	64 proc.
Time SP 'A'	1.75	1.02
L1 Misses SP 'A'	2.65	1.05
L2 Misses SP 'A'	0.93	0.79
Time SP 'B'	0.97	0.98
L1 Misses SP 'B'	0.97	1.00
L2 Misses SP 'B'	0.93	0.91
Time BT 'A'	0.96	0.95
L1 Misses BT 'A'	1.02	1.00
L2 Misses BT 'A'	0.95	0.96
Time BT 'B'	1.00	0.99
L1 Misses BT 'B'	0.98	1.00
L2 Misses BT 'B'	0.99	0.99

Table 10: Impact of array padding on SP and BT.

of L1 misses despite the fact that L2 data cache misses increase by 7% for 16 processors and by 26% for 64 processors. The SP benchmark uses data tiles that have one innermost non-distributed dimension; this causes logically contiguous elements in the other dimensions to be farther apart than would be the case if this local dimension did not exist. This increases the potential for cache conflicts between adjacent hyperplanes used in dimensional line sweeps. BT uses an outermost local dimension so it does not exhibit this problem.

For SP class B and BT, the results in table 10 show that using padding increases L1 and, in particular, L2 cache misses by having unused elements in the cache lines. This is not offset by a reduction in conflict misses.

Overall, these results show that dHPF needs a more sophisticated algorithm to determine the amount of padding needed for particular arrays instead of the simple heuristic we are currently using. In spite of this, dHPF's padding produces very good results for SP class A and does not incur heavy overhead in the other cases (slowdowns of no more than 5%), where it is ineffective.

7. Conclusions

The results presented in this paper show that the dHPF compiler is able to virtually match the performance of sophisticated, carefully tuned, hand-coded parallelizations of the NAS SP and BT benchmarks. To our knowledge this is the first time that data-parallel compilers have been able to deliver performance at this level for such tightly-coupled line sweep applications. Achieving this level of performance was not a matter of just implementing a few "big-ticket" optimizations. Delivering hand-coded performance with a data-parallel compiler requires a surprisingly broad spectrum of analysis and code generation techniques. With the exception of modest support for the multipartitioning data distribution, the optimizations we implemented in dHPF have broad applicability. We believe that they will be useful for other parallel architectures, although their relative importance may differ depending upon the parameters of the architecture.

Our experience is that everything affects scalability. Excellent parallel performance requires not only a good parallel algorithm, but also excellent resource utilization on the target parallel machine.

A fast, scalable program must make effective use of the processors, memory hierarchy and processor interconnect. For data parallel compilers, the implications are clear: discovering parallelism is only the beginning; exploiting it effectively is not necessarily as glamorous, but it is critically important. Optimizations must aim to effectively utilize all classes of resources in a parallel system. Only by targeting each potential source of inefficiency in compiler-generated parallel code can data-parallel compilers achieve the level of performance that will make them acceptable to application scientists.

Our ongoing work is focused on exploring data-parallel compiler issues for other types of numerical applications.

References

[1] High Performance Fortran Forum, "High Performance Fortran language specification," *Scientific Programming*, vol. 2, no. 1-2, pp. 1–170, 1993.

[2] G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works*. Morgan-Kaufmann, 1994.

[3] W. Gropp, M. Snir, B. Nitzberg, and E. Lusk, *MPI: The Complete Reference*. MIT Press, second ed., 1998.

[4] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi, "High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes," in *Proceedings of SC98: High Performance Computing and Networking*, (Orlando, FL), Nov 1998.

[5] V. Adve and J. Mellor-Crummey, "Using Integer Sets for Data-Parallel Program Analysis and Optimization," in *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, (Montreal, Canada), June 1998.

[6] J. Mellor-Crummey, V. Adve, B. Broom, D. C. a Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi, "Advanced optimization strategies in the Rice dHPF compiler," *Concurrency: Practice and Experience*, vol. 14, no. 8-9, pp. 741–767, 2002.

[7] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[8] N. Naik, V. Naik, and M. Nicoules, "Parallelization of a class of implicit finite-difference schemes in computational fluid dynamics," *International Journal of High Speed Computing*, vol. 5, no. 1, pp. 1–50, 1993.

[9] J. Bruno and P. Cappello, "Implementing the beam and warming method on the hypercube," in *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, (Pasadena, CA), pp. 1073–1087, Jan. 1988.

[10] J. Bruno and P. Cappello, "Implementing the 3D alternating direction method on the hypercube," *Journal of Parallel and Distributed Computing*, vol. 23, pp. 411–417, Dec. 1994.

[11] S. L. Johnsson, Y. Saad, and M. H. Schultz, "Alternating direction methods on multiprocessors," *SIAM Journal of Scientific and Statistical Computing*, vol. 8, no. 5, pp. 686–700, 1987.

[12] 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.

[13] 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.

[14] D. Chavarría-Miranda, J. Mellor-Crummey, and T. Sarang, "Data-parallel compiler support for multipartitioning," in *European Conference on Parallel Computing (Euro-Par)*, (Manchester, United Kingdom), Aug. 2001.

[15] A. Darte, D. Chavarría-Miranda, R. Fowler, and J. Mellor-Crummey, "Generalized multipartitioning for multi-dimensional arrays," in *Proceedings of the International Parallel and Distributed Processing Symposium*, (Fort Lauderdale, FL), Apr. 2002.

[16] Portland Group Inc., "PGI HPF versions of the NAS benchmarks." `ftp://ftp.pgroup.com/pub/HPF/examples/npb.tar.gz`, 1998.

[17] M. Frumkin, H. Yin, and J. Yan, "Implementation of nas parallel benchmarks in high performance fortran," in *Proceeding of the International Parallel and Distributed Processing Symposium*, (San Juan, Puerto Rico), Apr. 1999.

[18] M. Frumkin, M. Hribar, H. Yin, A. Waheed, and J. Yan, "A comparison of automatic parallelization tools/compilers on the sgi origin 2000," in *Proceedings of SC'98: High Performance Networking and Computing*, (Orlando, FL), Nov. 1998.

[19] K. McManus, M. Cross, S. Johnsson, and P. Leggett, "Issues and strategies in the parallelisation of unstructured multiphysics codes," in *Proceedings of Parallel and Distributed Computing for Computational Mechanics*, 1997.

[20] Y. Seo, H. Iwashita, H. Ohta, and S. Takahashi, "HPF/JA: HPF extensions for real-world parallel applications," in *Proceedings of the 2th Annual HPF User Group meeting*, (Porto, Portugal), June 1998.

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

[22] D. Chavarría-Miranda and J. Mellor-Crummey, "Towards compiler support for scalable parallelism," in *Proceedings of the Fifth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Lecture Notes in Computer Science 1915, (Rochester, NY), pp. 272–284, Springer-Verlag, May 2000.

[23] M. Gerndt, "Updating distributed variables in local computations," *Concurrency: Practice and Experience*, vol. 2, pp. 171–193, Sept. 1990.