Data-Parallel Compiler Support for Multipartitioning*

Daniel Chavarría-Miranda, John Mellor-Crummey, and Trushar Sarang

Dept. of Computer Science MS132, Rice University, Houston, TX 77005
{danich,johnmc}@cs.rice.edu

Abstract.

Multipartitioning is a skewed-cyclic block distribution that yields better parallel efficiency and scalability for line-sweep computations than traditional block partitionings. This paper describes extensions to the Rice dHPF compiler for High Performance Fortran that enable it to support multipartitioned data distributions and optimizations that enable dHPF to generate efficient multipartitioned code. We describe experiments applying these techniques to parallelize serial versions of the NAS SP and BT application benchmarks and show that the performance of the code generated by dHPF is approaching that of hand-coded parallelizations based on multipartitioning.

1 Introduction

High Performance Fortran (HPF) and OpenMP provide a narrow set of choices for data and computation partitioning. While their standard partitionings can yield good performance for loosely synchronous computations, they are problematic for more tightly-coupled computations such as line sweeps. Line sweep computations are the basis for Alternating Direction Implicit (ADI) integration—a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation [5, 10], as well as a variety of other computational methods [10]. Recurrences along each dimension of the data domain make this class of computations difficult to parallelize effectively.

To support effective parallelization of line-sweep computations, a sophisticated strategy for partitioning data and computation known as *multipartitioning* was developed [5, 10]. Multipartitioning distributes arrays of two or more dimensions among a set of processors so that for computations performing a line sweep along any one of the array's data dimensions, (1) all processors are active in each step of the computation, (2) load-balance is nearly perfect, and (3) only

* This work has been supported in part by NASA Grant NAG 2-1181, DARPA agreement number F30602-96-1-0159, and 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. 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 sponsoring agencies.

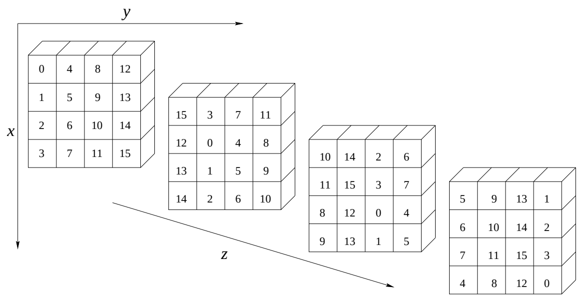

Fig. 1. 3D Multipartitioning on 16 processors.

a modest amount of coarse-grain communication is needed. These properties are achieved by carefully assigning each processor a balanced number of tiles between each pair of adjacent hyperplanes that are defined by the cuts along any partitioned data dimension. Figure 1 shows a 3D multipartitioning for 16 processors; the number in each tile indicates the processor that owns the block. For 3D problems, "diagonal" multi-partitionings [10] can be applied when $\sqrt{p}$ is integral, where p is the number of processors. This strategy involves partitioning the data domain into $p^{\frac{3}{2}}$ tiles. Each processor handles $\sqrt{p}$ tiles arranged along diagonals through the data domain. Recently, we developed an algorithm for applying multipartitionings efficiently on an arbitrary number of processors, which significantly broadens their applicability [7].

A study by van der Wijngaart [11] of implementation strategies for hand-coding parallelizations of Alternating Direction Implicit Integration (ADI) found that 3D multipartitioning was superior to both static block partitionings with wavefront parallelism, and dynamic block partitionings in which each phase of computation is perfectly parallel but data is transposed between phases.

Our earlier research on data-parallel compilation technology to support effective, semi-automatic compiler-based parallelizations of ADI line-sweep computations focused on exploiting wavefront parallelism with static block partitionings by using a coarse-grain pipelining strategy [1]. Although the performance we achieved with this approach was superior to that achieved for a version of the codes using dynamic block partitionings compiled with the Portland Group's *pghpf* compiler [4], both of the compiler-based parallelization strategies fell significantly short of the performance achieved by hand-coded parallelizations of the applications based on multipartitioning [1].

To closer approach the performance of hand-coded line-sweep computations with compiler-based parallelizations, we have been developing data-parallel compiler support for multipartitioning. A previous paper [6] describes basic compiler and runtime techniques necessary to support multipartitioned data distributions and a prototype implementation in the Rice dHPF compiler. Measurement of code generated with this early prototype identified several opportunities for improving performance. This paper describes the design and implementation of key techniques that address problems identified by reducing communication frequency, reducing message volume, and simplifying generated code.

Section 2 briefly introduces the Rice dHPF compiler and sketches its implementation of multipartitioning. Section 3 describes new optimizations that address performance issues with multipartitioning. Section 4 describes general optimizations that were crucial to generating efficient code for the complex iteration spaces that arose with multipartitioned code. Section 5 compares the performance of our compiler-based multipartitionings of the NAS SP and BT application benchmarks [3] (two codes that use ADI integration to solve the Navier-Stokes equation in three dimensions) and with the performance of hand-coded multipartitionings. Section 6 presents our conclusions.

2 The dHPF Compiler

The dHPF compiler [1, 2] translates HPF programs into single-program-multiple-data node programs in Fortran 77 that use MPI message-passing primitives for communication. dHPF is based on an abstract equational framework that expresses data parallel program analyses and optimizations in terms of operations on sets of integer tuples. These tuples represent data elements, processors and iterations [2]. The Omega Library [9, 8] serves as the implementation technology for the framework. Our equational framework has enabled us to implement simple, concise, and general formulations of analyses and optimizations. Because of the generality of these formulations, it has been possible to implement a comprehensive collection of advanced optimizations to support semi-automatic parallelization of scientific programs that are broadly applicable.

To support multipartitioned arrays, we extended dHPF to treat each data tile for a multipartitioned array as a block in a block partitioned array that is assigned to its own virtual processor and augmented code generation to map an appropriate collection of virtual processors to each physical processor [6]. Each data tile is extended using shadow regions to hold non-local data received from adjacent tiles. On each processor, all local tiles for a multipartitioned array are dynamically allocated as contiguous data. Storage is indexed in column-major order, where the leftmost dimensions are the original array dimensions and a new rightmost dimension corresponds to the *tile index*. All communication and computation for a tile is defined in terms of the data mapped to that tile.

Code generation for multipartitioned loop nests is a two step process. First we generate code for executing a loop nest for a single tile. Then, we wrap this code in a loop that iterates over all the tiles assigned to a physical processor. Communication generation for tiles is handles similarly. Communication pinned inside a computational loop by data dependences is handled for a tile as if its virtual processor is a physical processor. Communication operations that are vectorized outside of all computational loops over data dimensions are each enclosed in their own tile enumeration loop.

3 Multipartitioning Optimizations

From a processor's perspective, a multipartitioned computation is organized as computation on each of a series of tiles. To avoid unnecessary serialization between physical processors, each processor's tile computations must be scheduled in the proper order. To achieve good scalability, communication between processors must be as infrequent as possible. We describe optimizations that address these two issues for compiler-generated multipartitioned computations.

Tile Scheduling & Tile Loop Placement. A loop nest operating on multipartitioned data involves having each processor perform the portion of the loop's computation associated with each of its tiles. When communication is required to satisfy loop-carried data dependences between tiles, the tile enumeration order must be chosen carefully to avoid unnecessary serialization. This problem is unique to multipartitioning. Because of the way tiles are assigned to processors in multipartitioned distributions, choosing to iterate through a processor's tiles along any one multipartitioned data dimension determines the processor's iteration order along other multipartitioned dimensions as well. (This can be appreciated by considering a directional sweep over any processor's tiles in Fig. 1.)

If there are no loop-carried, processor-crossing true dependences that require communication, then any tile enumeration order is equally good. Otherwise, the data dimension along which the communication is flowing is selected as the tile dimension driving the enumeration order, and the tile iteration direction is selected to flow in the same direction as the communication. With dHPF's general computation partitioning model, we can always choose computation partitionings for statements in a loop nest so that all communication caused by loop-carried dependences in one data dimension flows in the same direction.

As long as processor-crossing dependences flow along only one dimension of multipartitioned array, any tile enumeration order that does not sweep in the opposite direction of the communication will be correct. However, full parallelism will be realized for a loop nest only if the tile dimension is the same as the dimension along which communication occurs, and the tile enumeration order is the same as the direction of the communication. If a multipartitioned computation involves communication along more than one dimension, the computation will be partially serialized regardless of tile enumeration order.

Communication Placement. In a loop nest, a processor-crossing true data dependence on a reference can be preserved by placing communication at the proper loop level. Loop-carried dependences can be preserved by placing communication within the loop carrying the dependence. Loop-independent dependences can be preserved by placing it within the loop that is the least common ancestor of the dependence source and sink.

Loop nests iterating over the data dimensions of a multipartitioned array possess the unique property that if there is only communication along a single direction of a single data dimension, it can be hoisted out of all enclosing loops over data dimensions without reducing parallelism. dHPF exploits this property to vectorize communication for processor crossing true dependences (both loop-carried and loop-independent) out of multipartitioned loops. This optimization enables loops to iterate over multipartitioned data in stride-1 order, regardless of where processor-crossing dependences exist. Without this optimization, communication would be pinned at some level in the loop nest, resulting in more messages of smaller size, which is more costly.

Communication placement in dHPF occurs in two steps. First, the compiler computes an initial placement in which data dependences are used to determine for each reference the loop level at which communication might be needed. This placement is safe for any data layout and any possible computation partitioning for each of the program's statements. Second, we apply a placement refinement algorithm that uses information about the data layout and computation partitionings selected to attempt to improve the initial placement. Placement refinement can hoist communication out of a loop, if it flows along the tile iteration dimension and direction and all statements in the loop nest are multipartitioned.

Aggregating Communication Across Tiles. A key property of multipartitionings is that a single physical processor owns all of the tiles that are neighbors of a particular processor's tiles along any given direction. For example, in Fig. 1 the right neighbor (if any) of each of processor 1's tiles along the y dimension belongs to processor 5. Thus, if a processor's tiles need to shift data to neighbors along a particular dimension, the processor needs to send values to only one other processor. For communication that has been vectorized completely out of loops over multipartitioned data dimensions, this property must be exploited to achieve scalable performance. Otherwise, in a 3D multipartitioning, a vectorized communication would require a processor to send $\sqrt{p}$ messages—one per tile.

To avoid this scaling problem, when the dHPF compiler generates code for each fully vectorized communication event, it sends data from all tiles of an array on the owning processor in a single message to the corresponding recipient. This optimization is a major factor in reducing communication frequency, although it doesn't reduce its volume.

4 General Optimizations

Our quest to achieve hand-coded performance with multipartitioned code generated by dHPF led us to devise several new types of optimizations that are universally applicable. Most of these optimizations are refinements of dHPF's analysis and code generation strategies for constructing and manipulating sets of integer tuples using the Omega Library [8]. The set-based optimizations we describe here improve the ability of dHPF to analyze and optimize set representations that arise with complex programs.

Formulating Compact Sets. The Omega Library has enabled us to develop very general formulations of sophisticated optimizations in the dHPF compiler. However, sophisticated optimizations such as partially replicating computation to reduce communication can give rise to unnecessarily complex sets. Complex sets can cause Omega Library set manipulation operations to exhaust resources or generate inefficient code. Analysis of sets that arose during compilation showed that disjunctions of sets in which some tuple dimensions were drawn from a range of adjacent constants were not being collapsed together. We discovered that sets with compact ranges of constant terms could be collapsed by recomputing a set S as $\texttt{ConvexHull}(S) - (\texttt{ConvexHull}(S) - S)$. This process forces the set to be recomputed as a difference from a single conjunct, which has the effect of reducing the set to its most compact form. This optimization enabled us to partially-replicate computation for some unrolled code and save a factor of 5 in communication volume as described in the experiments.

Communication Factoring. As a consequence of partially-replicated computation partitionings, a single reference can cause communication in multiple directions. With multipartitioned or block distributions, partial replication of computation around block boundaries will cause right-hand-side array references to require communication in both directions along each partitioned array dimension. While the integer tuple representation of communication sets used in dHPF can represent nearest-neighbor communication to each of a block's neighbors, the code generated for such communication is inefficient because for each neighbor, it must check which face of the non-convex communication set is needed. By factoring the communication event into simpler communication events—in this case, a separate one for each data dimension and each communication direction—the resulting communication is more efficient and the generated code is much shorter. After factoring the communication, each communication typically is represented by a simple convex set.

Communication Set Construction. The form of set representations based on the Omega Library is largely determined by how the sets are constructed. Carefully constructing sets yields equivalent but simpler forms, speeds up analysis and code generation, and produces cleaner and simpler output. To simplify the representation of communication sets, we group data references contributing to a communication set into equivalence classes. Equivalence classes are formed by inspecting the value number of each subscript expression in the communication-causing dimension and grouping conformant references together. References with different value numbers in partitioned dimensions should be grouped into different equivalence classes. Each equivalence class has a simple convex representation. The final communication set is constructed by unioning together sets constructed for each equivalence class.

Communication Coalescing. Communication is initially scheduled independently for each data reference. Then, multi-directional communication for any single reference (caused by partially-replicated computation partitionings) is factored into separate operations. At this point, there may exist many communication events that send and receive overlapping sets of data. To avoid redundant communication, we merge conformant communication events moving overlapping sets of data. Since communication events draw their representation from data references and computation partitionings, comformance between two communication events is not easily determined. For example, the data reference `a(i,j,k)` with a computation partitioning of `ON_HOME a(i+1,j,k)` causes communication for the last row of data in each tile. The data reference `a(i-1,j,k)` with a partitioning of `ON_HOME a(i,j,k)` requires exactly the same data to be communicated. To detect such redundancy, we convert them into a normal form by removing offsets in `ON_HOME` references and adjusting data references accordingly.

General Communication Aggregation. Rather than sending separate messages for each distributed array to be communicated, we extended the dHPF compiler to combine messages at the same point in the code that are addressed to the same recipient from a particular sender into a single message. Two single logical shift communication events for different distributed arrays may be safely aggregated if both arrays are mapped to the same HPF template, and communication flows in one direction along the same dimension of the template.

Code Generation. Generating SPMD code templates (that contain appropriately bounds-reduced loops and conditionals which partition the computation among the available processors) for vectors of disjunctive iteration spaces is a complex problem we encounter when generating code for iteration spaces that have been partially-replicated along data partitioning boundaries.

While Omega contains a sophisticated procedure for generating code templates for complex iteration spaces [9], we found that applying Omega's code generation algorithm to vectors of disjunctive iteration spaces often produced inefficient code templates in which a set of guards terms is repeatedly tested and placeholders for code fragments may repeat many times. In the dHPF compiler, we take several steps to generate high-quality code for such complex iteration spaces. First, we exploit context information present at the enclosing scope to avoid testing conditions already known to be true. Second, when generating code for a vector of iteration sets, we avoid repeatedly testing the same conditions by (a) projecting out satisfiability constraints that are common to all of the iteration sets and test them only once, and (b) refining Omega's code generation algorithm for vectors of iteration spaces to merge adjacent statements subject to the same guards. Third, to avoid unnecessarily complex guards and code replication, we merge together adjacent instances of the same statement and simplify guards from disjoint disjunctive form to simple disjunctions. Finally, to avoid enforcing the same constraints twice, we project away satisfiability constraints from guards around a loop nest that are enforced by the loop bounds.

Generating Optimizable Code. The dHPF compiler translates an HPF program into SPMD Fortran 77 code with MPI communication. The generated code will run fast only if the target system's Fortran compiler can generate good machine code for it. Our experimentation with multipartitioned code generated by dHPF has been performed on an SGI Origin 2000 with MIPS R10000 processors using SGI's MIPSpro Fortran compiler. Achieving good cache utilization on this architecture is essential for good performance. The two key issues that had to be addressed were *avoiding conflict misses*, and *exploiting software prefetching*. We avoid conflict misses that arise with dynamically-allocated tiles in multipartitioned arrays by padding each tile dimension to an odd length. Since we depend on the target system's Fortran compiler to insert software prefetches to hide the latency of memory accesses, the multipartitioned Fortran code that dHPF generates must be readily analyzable. We had to adjust our generated code so the backend compiler did not erroneously infer aliasing between separate arrays, and we had to avoid product terms in subscripts (including those that might arise by forward substitution). One significant change this required was using Cray pointer notation for accessing multi-dimensional dynamically-allocated arrays rather than using linearized subscripts.

5 Experimental Results

We applied the dHPF compiler to generate optimized multipartitioned parallelizations of serial versions (NPB2.3-serial release) of the NAS BT and SP application benchmark codes [3] developed by NASA Ames. BT and SP solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. While BT solves block-tridiagonal systems of 5x5 blocks, SP solves scalar penta-diagonal systems. Both codes are iterative computations. In each time step, the codes calculate boundary conditions and then compute the right hand sides of the equations. Next, they solve banded systems in computationally-intensive bi-directional sweeps along each of the 3 spatial dimensions. Finally, they update flow variables. Parallel versions of these computations require communication to compute the right hand sides and during the forward and backward line sweeps.

We lightly modified the serial versions of SP and BT for use with dHPF. We added HPF data distribution directives, HPF INDEPENDENT and NEW directives (for array privatization), and a few one trip loops (around a series of loops to facilitate fusing communication). Multipartitioning is specified using a new MULTI distribution keyword for template dimensions. Our experiments were performed on an SGI Origin 2000 node of ASCI Nirvana (128 250MHz R10000, 32KB (I)/32KB (D) L1, 4MB L2 (unified)).

5.1 NAS BT

Effectively parallelizing the NPB2.3-serial version of BT with dHPF required a broad spectrum of analysis and code generation techniques. Multipartitioning and new optimizations in dHPF greatly improved both the performance and scalability of the generated code compared to our earlier parallelizations [1] based on block partitionings. Both the sequential and parallel performance of dHPF's compiler-multipartitioned code is much closer to the hand-coded version. Figure 2 shows a 16-processor parallel execution trace for one steady-state iteration of our compiler-generated multipartitioned code for the class A (64^3) problem size. This parallelization is quite similar to the hand-coded multipartitioning.

Using non-owner computes computation partitionings to partially replicate computation along multipartitioned tile boundaries led to dramatic reductions of communication volume. In BT's `compute_rhs` subroutine, partially replicating the computation of the privatizable temporary arrays `rho_q`, `qs`, `us`, `vs`, `ws`, and `square` along the boundaries of a multipartitioned tile avoided communication of these six variables. No additional communication was needed to partially-replicate this computation because the boundary planes of the multipartitioned `u` array needed by the replicated computation were already being communicated. This optimization cut the communication volume of `compute_rhs` by nearly half. In BT's `lhsx`, `lhsy`, and `lhsz` subroutines, partially replicating computation along the partitioning boundaries of two arrays, `fjac` and `njac`, whose global

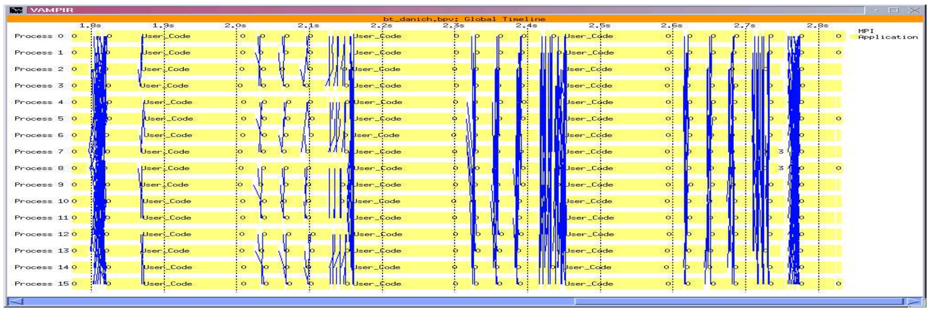

Fig. 2. dHPF-generated NAS BT using 3D multipartitioning.

# CPUs	hand-coded	dHPF	% diff.
1	1.06	1.09	-2.64
4	3.28	3.34	-1.77
9	7.73	7.26	6.14
16	14.21	13.49	5.10
25	21.08	20.66	1.98
36	29.78	28.77	3.40
49	39.73	33.73	15.10
64	48.13	51.15	-6.28

Class A: $64 \times 64 \times 64$

# CPUs	hand-coded	dHPF	% diff.
1	0.98	0.92	5.85
4	3.37	2.91	13.48
9	4.91	5.63	-14.70
16	12.30	12.83	-4.34
25	19.09	19.91	-4.30
36	30.95	28.80	6.93
49	52.82	37.04	29.88
64	66.04	47.03	28.78
81	82.28	53.57	34.89

Class B: $102 \times 102 \times 102$

Fig. 3. Comparison of hand-coded and dHPF speedups for NAS BT.

dimensions are `(5,5,IMAX,JMAX,KMAX)`, saved a factor of five in communication. Rather than communicating planes of computed values for these arrays across partitions in the `i`, `j`, and `k` dimensions, we communicated sections of `rhs(5,IMAX,JMAX,KMAX)`, which is a factor of five smaller.

Figure 3 compares speedups of NASA Ames' hand-coded parallelization using multipartitioning with those of our dHPF-generated code. All speedups are relative to the performance of the original sequential code. The columns labeled "% diff" show the differences between the speedup of the dHPF-generated code and the speedup of the hand-coded version, relative to the speedup of the hand-coded version. The good parallel performance and scalability of the compiler-generated code comes from applying multipartitioning in conjunction with partially-replicated computation to reduce communication, aggressive communication placement, and aggregating communication across both tiles and different arrays. A measure of the effectiveness of these communication optimizations is that for a 16-processor class A execution, dHPF had only 1.5% higher communication volume, and 20% higher message frequency than the hand-coded implementation.

While the performance of dHPF-generated code closely tracks that of the hand-coded version for the 64^3 problem size, for the 102^3 size the parallel efficiency of dHPF's generated code begins to lag at 49 processors. As the number of

# CPUs	hand-coded	dHPF	% diff.
1	1.01	0.96	5.50
4	4.21	3.29	21.83
9	11.60	7.75	33.24
16	16.21	14.43	10.98
25	21.00	22.68	-7.98
36	30.69	28.44	7.31
49	42.43	30.89	27.19
64	67.57	35.15	32.42

Class A: $64 \times 64 \times 64$

# CPUs	hand-coded	dHPF	% diff.
1	0.80	0.78	2.67
4	2.86	2.52	12.13
9	7.74	6.17	20.26
16	13.01	11.36	12.63
25	22.15	17.77	19.75
36	36.52	25.72	29.57
49	51.78	33.22	35.85
64	58.35	41.52	28.84
81	74.95	45.52	39.26

Class B: $102 \times 102 \times 102$

Fig. 4. Comparison of hand-coded and dHPF speedups for NAS SP.

processors increases, the surface-to-volume ratio of multipartitioned tiles grows proportional to $\sqrt{p}$ and dHPF's packing and unpacking of data in array overlap regions causes an increase in secondary data cache misses. This degrades the performance of communication-sensitive line sweeps the most. The hand-coded implementation uses data in communication buffers without unpacking.

5.2 NAS SP

Effectively parallelizing SP also required a broad spectrum of optimizations as with BT. Despite the fact that the dynamic communication patterns of dHPF's multipartitioned parallelization resemble those of the hand-coded parallelization, there is still a performance gap between the two implementations. Figure 4 compares speedups of NASA Ames' hand-coded parallelization using multipartitioning with those of our dHPF-generated code. All speedups are relative to the performance of the sequential code for the respective NPB2.3-serial distribution. Most of the difference comes from extra communication volume present in the compiler-generated code for SP's `lhs<xyz>` routines.

Currently, the dHPF compiler uses a procedure-local communication placement analysis. This approach schedules communication in each procedure even though the values might already be available locally. Interprocedural communication analysis and placement analysis would be needed to eliminate this additional communication. We measured a significant increase in secondary cache misses in the sweep routines caused by how dHPF manages communication buffers and unpacks non-local data into overlap regions. For `z_solve`, the overhead was 100%. This cost could be reduced by accessing remote data directly from the packed buffers, but would prove challenging to implement due to the non-contiguous nature of the data.

For SP, non-local values gathered by the `compute_rhs` routine cover the non-local values needed by the `lhsx`, `lhsy` and `lhsz` routines. In a 16-processor execution for a class A problem size, this unnecessary communication in `lhsx`, `lhsy` and `lhsz` causes the communication volume of our dHPF-generated code to be 14.5% higher than for the hand-coded parallelization. Although the additional volume is modest, since this extra communication is forward and backward along each of the spatial dimensions, the communication frequency of the dHPF-generated code to be 74% higher than the hand-coded parallelization.

Like with BT, partially-replicating computation at the boundaries of multipartitioned tiles offered significant benefits for SP. In SP's `lhsx`, `lhsy`, and `lhsz` routines, replicating computation of boundary values of `cv`, a partitioned 1-dimensional vector aligned with a multipartitioned template, eliminated the need to communicate these boundary values at loop depth two between their definition and use. Although partially replicating computation of `cv` required communicating two additional planes of the three-dimensional multipartitioned array `us` in each of these routines, this communication was fully vectorizable, whereas the communication of `cv` we avoided was not. In SP's `x_solve`, `y_solve`, and `z_solve` routines, dHPF's ability to vectorize loop-carried communication out of the entire loop nest over a multipartitioned tile was the key to achieving one message per tile.

6 Conclusions

Van der Wijngaart showed that multipartitioning yields superior parallel performance for tightly-coupled line sweep computations than other partitioning choices [11]. In the dHPF compiler, we successfully implemented support for multipartitioning. We were able to apply aggressive optimizations in dHPF in the complicated circumstances arising in multipartitioned code principally because our compiler optimizations are formulated in a very general way—as manipulation of sets of integer tuples [2]. Only the combination of multipartitioning and aggressive optimizations enabled us to approach hand-coded performance on the NAS SP and BT benchmarks. Achieving high performance with compiler-generated parallelizations requires paying attention to *all* the details. Multipartitioning by itself leads to balanced computation, but matching the hand-coded communication frequency and volume, and also the scalar performance is necessary to achieve competitive performance.

Remaining differences in performance between the dHPF-generated code and the hand-coded parallelizations result from three factors. First, dHPF-generated code incurs higher data-movement overheads in its communication support code than the hand-coded implementations, which carefully reuse buffers and use communicated data directly out of buffers rather than unpacking. Second, dHPF-generated code contains extra communication that comes from a lack of interprocedural analysis and placement. Third, the hand-coded implementation hides communication latency by scheduling local computation while data is in flight. Since in the serial version of the code analyzed by dHPF, the overlapped local computation is in another routine, achieving this optimization would require more global analysis and transformation.

Our experiments with two line-sweep codes show that our compiler technology is able to simultaneously optimize all aspects of program performance. We have demonstrated that it is possible to use data-parallel compiler technology to effectively parallelize tightly-coupled line-sweep applications and yield performance that is approaching that of sophisticated hand-coded parallel versions.

Acknowledgments: We thank Vikram Adve for his involvement in the early design and implementation discussions of this work.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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*, pages 1073–1087, Pasadena, CA, Jan. 1988.
6. 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, pages 272–284, Rochester, NY, May 2000. Springer-Verlag.
7. A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. On efficient parallelization of line-sweep computations. In *9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, June 2001.
8. 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.
9. 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.
10. 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*, 5(1):1–50, 1993.
11. R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Proceedings of Supercomputing 1993*, pages 102–111. IEEE Computer Society Press, 1993.