CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **1**:1–20 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Advanced Optimization Strategies in the Rice dHPF Compiler

J. Mellor-Crummey[1,*], V. Adve[2], B. Broom[1], D. Chavarría-Miranda[1], R. Fowler[1], G. Jin[1], K. Kennedy[1] and Q. Yi[1]

[1] *Department of Computer Science—MS 132, Rice University, 6100 Main St., Houston, TX 77005*
[2] *Computer Science Department—MC-258, University of Illinois at Urbana-Champaign, 1304 West Springfield Ave., Urbana, IL 61801.*

SUMMARY

High Performance Fortran (HPF) was envisioned as a vehicle for modernizing legacy Fortran codes to achieve scalable parallel performance. To a large extent, today's commercially available HPF compilers have failed to deliver scalable parallel performance for a broad spectrum of applications because of insufficiently powerful compiler analysis and optimization. Substantial restructuring and hand-optimization can be required to achieve acceptable performance with an HPF port of an existing Fortran application, even for regular data-parallel applications. A key goal of the Rice dHPF **compiler project has been to develop optimization techniques that enable a wide range of existing scientific applications to be ported easily to efficient HPF with minimal restructuring. This paper describes the challenges to effective parallelization presented by complex (but regular) data-parallel applications, and then describes how the novel analysis and optimization technologies in the** dHPF **compiler address these challenges effectively, without major rewriting of the applications. We illustrate the techniques by describing their use for parallelizing the NAS SP and BT benchmarks. The** dHPF **compiler generates multipartitioned parallelizations of these codes that are approaching the scalability and efficiency of sophisticated hand-coded parallelizations.**

KEY WORDS: High Performance Fortran, Automatic Parallelization, Computation Partitioning, NAS Benchmarks, Multipartitioning

*Correspondence to: J. Mellor-Crummey, Department of Computer Science—MS 132, Rice University, 6100 Main St., Houston, TX 77005. E-mail: johnmc@cs.rice.edu.
Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-9120008
Contract/grant sponsor: NASA-AMES; contract/grant number: NAG 2-1181
Contract/grant sponsor: Los Alamos National Laboratory; contract/grant number: 03891-99-23
Contract/grant sponsor: DARPA and Rome Laboratory, Air Force Materiel Command, USAF†; contract/grant number: F30602-96-1-0159
Contract/grant sponsor: National Science Foundation; contract/grant number: ACI-9619020

1. INTRODUCTION

High Performance Fortran (HPF) [1, 2] was designed in the early 1990s with the intent of harnessing the power of automatic parallelization technology to provide a commercially viable high-level programming model for developing portable parallel programs. HPF provides an attractive model for parallel programming because of its relative simplicity. Principally, programmers write a single-threaded Fortran program and augment it with layout directives to map data elements onto an array of processors. HPF compilers use these directives to partition a program's computation among processors and to synthesize code for required data movement and synchronization. As of 1997, fifteen companies were offering HPF compilers, including all major parallel computer vendors, and nearly forty major applications had been developed in the language, some over 100,000 lines.

Despite considerable initial enthusiasm for HPF, it has not achieved widespread acceptance by scientists as the model of choice for developing parallel applications. The success of HPF has been principally limited by the shortcomings of its compilers, which were adapted from technology for automatic parallelizers of the late 80s and early 90s. This technology has not been sophisticated enough to deliver scalable performance across a variety of architectures and applications. Compilation techniques in use by commercial HPF compilers fail to generate code that achieves performance competitive with that of hand-coded programs for all but the most straightforward applications. As a result, application developers have been reluctant to use HPF. Some have chosen to hand-code applications using message-passing instead, while many others have simply delayed the transition to scalable parallelism.

It has been difficult for developers to use HPF compilers to parallelize existing codes. Commercial HPF compilers lack analysis and code generation capabilities to parallelize a wide range of loops written in a Fortran 77 style effectively. Therefore, for codes to achieve reasonable parallelism when compiled with commercial HPF compilers, they have had to be rewritten substantially. For example, the Portland Group developed HPF versions [3] of the NAS parallel application benchmarks [4] which show reasonable performance and scalability when compared with hand-coded message-passing (MPI) versions of the same benchmarks, coming within about a factor of 2. These results, however, were obtained by almost completely rewriting the benchmark source codes as part of the conversion to HPF. Key changes included selective unrolling of loops, inlining of procedures, and repeated data transposition to avoid wavefront (pipelined) communication. In addition, forward substitutions and loop realignments were necessary to avoid using privatizable arrays. The resulting codes were nearly twice the length of original serial versions of the benchmarks.

In addition, the restricted set of data partitionings allowed in HPF has turned out to be a significant limitation, particularly for tightly-coupled codes. For example, hand-written parallelizations of the NAS SP and BT application benchmarks use multipartitioning—a skewed-cyclic block distribution—that is not available in standard HPF, but which delivers exceptional performance and scalability for these applications. Alternative parallelizations based on either static or dynamic block distributions, which are supported by HPF, don't yield comparable efficiency, even in hand-coded implementations [5]. Ideally, HPF compiler technology should be extensible to support new user-specified distribution strategies.

Prepared using cpeauth.cls

A principal goal of the Rice dHPF compiler project has been to develop compiler optimization techniques necessary to deliver scalable parallel performance for a broad class of data-parallel applications. In particular, an aim of the project has been to develop compiler techniques that simplify the conversion of well-written codes into efficient HPF, and to preserve the benefits of any code restructuring a user may have done to improve memory hierarchy performance.

In this paper, we describe the challenges to effective parallelization presented by complex (but regular) data-parallel applications. We then describe how novel analysis and optimization technologies can be used to address these challenges. The emphasis in this paper is on describing the impact of the optimizations. The compiler analysis techniques and algorithms used to perform the optimizations are described elsewhere [6, 7, 8, 9, 10, 11, 12].

The next section describes some of the key challenging features of scientific programs. Subsequent sections describe the analysis and optimization techniques in dHPF, including computation partitioning optimizations (Section 3) and sophisticated data distributions (Section 4). Section 5 briefly mentions other optimizations that are essential to achieving high performance. Section 6 describes some experiments that demonstrate the effectiveness of these techniques for parallelizing lightly modified serial versions of two tightly coupled line sweep applications. The final section summarizes the key points of the paper and describes areas that need further research.

2. PERFORMANCE CHALLENGES

Most traditional HPF compilers use simple partitioning schemes for data and computation. The HPF directives determine the data partition, and the computation partition (`CP`), the assignment of computations to processors, is driven by some form of the "owner computes rule" heuristic [13]. Although some serial codes can be parallelized using this approach, others require more sophisticated approaches if HPF is to approach the performance of hand-coded parallel implementations. In this section, we describe several aspects of programs that make it difficult for HPF compilers to generate code that yields scalable performance competitive with hand-coded implementations. Here, and in the rest of the paper, we principally use examples drawn from serial versions (NPB2.3-serial release) of the NAS application benchmarks BT and SP [4]. The serial versions of these two codes were created by NASA scientists from hand-coded MPI implementations, so they embody parallelizable algorithms in near parallel form. BT and SP are both computational fluid dynamics codes that solve a discretized version of the Navier-Stokes equations in three dimensions. The main difference between them is that BT solves block-tridiagonal systems, whereas SP solves scalar penta-diagonal systems.

2.1. Temporary Arrays

Within complex loop nests in scientific codes, temporary arrays are often used to hold intermediate data values so that they can be reused in subsequent iterations or loop nests. However, when parallelizing code that uses temporary arrays, particular care must be taken to avoid compromising the performance and scalability of the application by replicating work excessively, introducing frequent communication, communicating too much data, or reducing node performance.

Consider the loop nest shown in Figure 1, which is from the `lhsy` subroutine of the NAS SP application benchmark. The temporary array `rhoq` is defined and used exclusively in this loop nest.

Prepared using cpeauth.cls

```
      do 10 k = 1, grid_points(3)-2
         do 10 i = 1, grid_points(1)-2
            do 20 j = 0, grid_points(2)-1
               ru1 = c3c4*rho_i(i,j,k)
               cv(j) = us(i,j,k)
20             rhoq(j) = dmax1(dy2+con43*ru1,dy5+c1c5*ru1,dymax+ru1,dy1)
            do 30 j = 1, grid_points(2)-2
               lhs(i,j,k,1) =   0.0d0
               lhs(i,j,k,2) = - dtty2 * cv(j-1) - dtty1 * rhoq(j-1)
               lhs(i,j,k,3) =   1.0d0 + c2dtty1 * rhoq(j)
               lhs(i,j,k,4) =   dtty2 * cv(j+1) - dtty1 * rhoq(j+1)
30             lhs(i,j,k,5) =   0.0d0
10       continue
```

Figure 1. Loop nest from subroutine `lhsy` of the NAS SP computational fluid dynamics benchmark.

Without loss of generality, we may assume that the data and computation of the `j` dimension of the `lhs` array is partitioned because there are analogous loop nests that operate along the `i` and `k` dimensions of the `lhs` array as well. Each array element `rhoq(j)` defined in the first `j` loop is subsequently used three times, by iterations `j-1`, `j`, and `j+1` of the second `j` loop. How are we to ensure that the required values for `rhoq` are available to the processors executing iterations of the second loop? One approach would be to replicate the entire computation of `rhoq` onto each of the processors; however, this does not yield scalable performance. A second approach would be to compute each element of `rhoq` on a single processor and to communicate elements as necessary between the definition and use. This would require two single-element communications between each pair of neighboring processors on each iteration of the `i` loop. On most architectures, this frequent synchronization would be costly. A third approach would be to expand `rhoq` to a three-dimensional temporary array and to distribute the `i` and `k` loops. In this case, each processor could compute a slab of the `rhoq` array for the range of the `j` dimension that it owns. A pair of communications to exchange planes of the `rhoq` array with neighbors along the partitioned `j` dimension would give each processor the values it needs to perform the computation of the second `j` loop. While this approach would avoid the excessive replicated work and communication of the previously described approaches, it would use considerably more storage for `rhoq` and decrease the temporal locality of accesses to `rhoq`. Both of these changes would reduce the cache utilization and diminish node performance of the parallelized code. Clearly, none of these approaches are ideal.

Temporary arrays can also complicate parallelization by causing excessive communication volume when using simple partitionings of data and computation. Consider the schematic shown in Figure 2 of a pair of loop nests from the `lhsy` subroutine in the NAS BT application benchmark. For both clarity and conciseness, we have abstracted away some of the complexity of the actual loop nests by presenting them in terms of loops over `l` and `m` and replacing statement bodies by calls to fictitious functions that are parameterized by `l` and `m`. In the actual code, the `l` and `m` loops are fully unrolled, and calls to the functions `f1`, `f2`, and `f3` are fully inlined (the code in these functions does not add to the

```
      do 10 k = 1, grid_points(3)-2
         do 10 j = 0, grid_points(2)-1
            do 10 i = 1, grid_points(1)-2
               do 10 l = 1, 5
                  do 10 m = 1, 5
                     fjac(l,m,i,j,k) = f1(l,m,u(1:5,i,j,k))
10                   njac(l,m,i,j,k) = f2(l,m,u(1:5,i,j,k))
      do 20 k = 1, grid_points(3)-2
         do 20 j = 1, grid_points(2)-2
            do 20 i = 1, grid_points(1)-2
               do 20 l = 1, 5
                  do 20 m = 1, 5
20                   lhs(l,m,i,j,k) =
     >                  f3(l,m,fjac(l,m,i,j-1:j+1,k),njac(l,m,i,j-1:j+1,k))
```

Figure 2. Schematic loop nest from subroutine `lhsy` of the NAS BT computational fluid dynamics benchmark.

communication cost or other parallelization overheads). As in the previous example, we may assume that the `j` dimension of the `lhs` array is partitioned, because there are analogous computations along the `i` and `k` dimensions as well. The first loop nest computes values for `njac` and `fjac`, two temporary arrays that are then used by the second loop nest. For each `ijk` triple, this pair of temporary arrays contains a total of 50 values for every five of the original `u` array from which they were computed. If the elements of `fjac` and `njac` are computed in parallel (each on a single processor), the boundary volumes of `fjac` and `njac` would have to be exchanged between neighboring processors. This results in a communication volume ten times larger than the best alternative strategy, which is to replicate the computation of those elements that are needed on multiple processors, requiring only the exchange of the boundary planes of `u` to compute those elements.

Problems similar to the two cases described above arise if all of the elements of an array are defined and used in one region or loop and there are later uses of the array elsewhere in the program. For example, consider the code in Figure 3 from subroutine `compute_rhs` of the NAS SP benchmark. In `compute_rhs`, `rhs` is evaluated along each of the x, y, and z directions. Along each direction, flux differences are computed and then adjusted by adding fourth-order dissipation terms. To reduce the number of operations, especially floating-point divisions, reciprocals—`rho_i`, `us`, `vs`, `ws`, `square`, and `qs`—are computed once, and are then used repeatedly as multipliers. Since the usage patterns of these variables are similar, we focus on a single reciprocal variable `rho_i`. Without partial replication of computation, the best CP choice for the statements define and use `rho_i` is the one chosen by the owner-computes rule. These CP choices, however, would cause the references `rho_i(i,j-1,k)` and `rho_i(i,j+1,k)` to become non-local. Similar references in the `rhs` computation along the z direction become non-local as well if the `k` dimension is partitioned. In this case, the boundary data of `rho_i` as well as the other five arrays would need to be communicated.

```
do k = 0, N-1; do j = 0, N-1; do i = 0, N-1
  rho_inv = 1 / u(i,j,k,1)
  rho_i(i,j,k) = rho_inv
  us(i,j,k) = u(i,j,k,2) * rho_inv
  vs(i,j,k) = u(i,j,k,3) * rho_inv
  ws(i,j,k) = u(i,j,k,4) * rho_inv
  square(i,j,k) = f( u(i,j,k,2:4) * rho_inv )
  qs(i,j,k) = square(i,j,k) * rho_inv

! x-direction ...

do k = 1, N-2 ;  do j = 1, N-2;  do i = 1, N-2      ! y-direction
  rhs(i,j,k,1) = ... u(i,j+1,k)      + ... u(i,j-1,k) ...
  rhs(i,j,k,2) = ... us(i,j+1,k)     + ... us(i,j-1,k) ...
  rhs(i,j,k,3) = ... square(i,j+1,k) + ... square(i,j-1,k) ...
  rhs(i,j,k,4) = ... ws(i,j+1,k)     + ... ws(i,j-1,k) ...
  rhs(i,j,k,5) = ... qs(i,j+1,k)     + ... qs(i,j-1,k)...
                 ... + rho_i(i,j+1,k) + ... + rho_i(i,j-1,k)
  ...

! z-direction ...
```

Figure 3. Loop from NAS SP subroutine `compute_rhs` with high communication for long-lived arrays.

2.2. Partitioning Complex Loop Nests

Scientific codes that have been carefully tuned for single-processor performance often have substantial intra-loop reuse of array values, both within and across iterations. Such reuse results in loop-independent and loop-carried data dependences. If two statement instances connected by a dependence are assigned to different processors, communication could be induced in an inner loop at considerable cost. Figure 4 shows a loop from subroutine `y_solve` of the NAS SP benchmark that illustrates the problem. In this loop nest, each loop-independent dependence between a pair of references is shown with a dashed arrow and each loop-carried dependence is shown with a solid arrow. Since there are analogous loops with loop-carried dependences along the `i` and `k` dimensions, without loss of generality we can assume that the `j` dimension is partitioned. Such loops pose a significant parallelization challenge for HPF compilers. Simply using an owner-computes partitioning would result in expensive communication inside the `i` loop to satisfy processor-crossing loop-independent dependences between processors managing different partitions along the `j` dimension. Loop distribution is the most common technique used to avoid communication inside loop nests in such cases; however, loop distribution must be applied sparingly because it reduces temporal reuse, which can increase cache misses and degrade performance. In this case, although some loop distribution can be applied, six of the ten assignment statements must stay together in the same loop, since they form a strongly-connected component of the dependence graph. Without careful partitioning and communication placement, the loop-independent dependences among these statements require conditional communication at the `i` level.

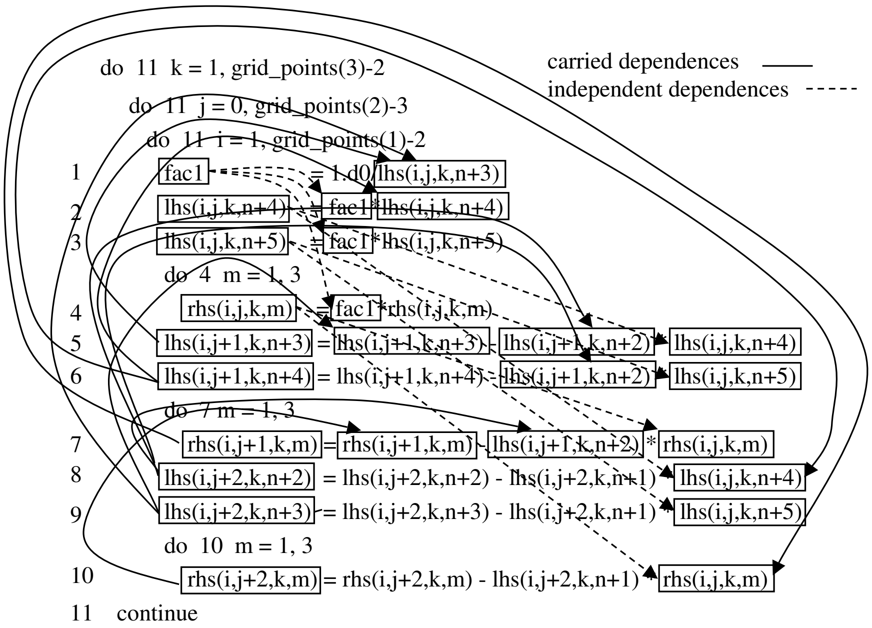

Figure 4. Loop from NAS SP subroutine `y_solve` with complex dependences.

2.3. Parallelizing Tightly-coupled Computations

While HPF compilers can readily yield good performance for codes that are embarrassingly parallel (no communication) or loosely synchronous (communication only between loop nests), tightly coupled codes with loop-carried dependences are difficult to parallelize effectively. An important class of computations that requires such tight coupling consists of solvers for partial-differential equations based on Alternating-Direction Implicit (ADI) integration. This technique has been widely used in computational fluid-dynamics codes for solving the Navier-Stokes equation [14, 15, 16]. Line-sweep computations form the heart of ADI integration. Figure 4 shows a loop nest that is part of a solver kernel that performs line-sweep computation. As described in the previous section, such computations are difficult to parallelize effectively and they are not well-suited to the standard block partitionings supported by HPF. The two reasonable strategies of parallelization possible with HPF are (1) static block partitionings, resulting in wavefront parallelism, and (2) dynamic block partitionings, resulting in fully parallel computation phases separated by data transpose operations on the partitioned arrays. A study by van der Wijngaart using hand-coded parallelizations of ADI integration showed that both these strategies yield inferior performance compared to a sophisticated skewed-cyclic block distribution known as multipartitioning [5]. This distribution and methods for compiling for it are described further in Section 4.

2.4. Computing with Very Large Arrays

Finally, another challenge concerns computing with very large arrays that exceed a computer's available physical memory. Clearly, for nodes without virtual memory, an application must input, process, and output such arrays in sections. Moreover, even when virtual memory is available, using it to manage very large arrays in technical application does not yield particularly good performance. Processing arrays in sections is almost always better.

There are three reasons virtual memory is not particularly well suited to scientific computations on very large arrays. First, virtual memory page replacement policies that are variants of least-recently-used replacement are poor matches for scientific computations that process different elements of a large array in each iteration of some loop. Such applications result in little (if any) temporal reuse of cached data. Consequently, reserving large amounts of a node's memory to support such an application will not yield a significant performance improvement; however, it will force data out of memory for other applications running on the node and will significantly degrade their performance [17]. Second, since application data managed in virtual memory are not fetched until the application references the data, the application will stall any time it references a data value that it has not referenced in the very recent past. Operating system prefetching may not help avoid such stalls if there is a mismatch between the layout of the array and the order of the computation. Finally, the typical virtual memory replacement unit is a few pages representing a small number of disk blocks; large I/O requests of multiple megabytes can be handled much more efficiently. Consequently, applications that explicitly manage their memory resources can have significantly higher performance, sometimes by a factor of one hundred [18].

For HPF to be widely useful, compilers must generate high-performance code for complex cases such as those described in this section. It is critical that HPF compilers handle complex cases well because it is in precisely such cases that users would have difficulty parallelizing codes by hand and would want to draw upon compiler technology for assistance. The following sections describe some of the advanced analyses and optimizations used by the dHPF compiler to produce parallel code whose efficiency is approaching that of highly tuned, hand-written parallel programs. First, in Section 3, we describe computation partitioning optimizations that help dHPF to effectively parallelize programs that use temporary arrays or having complex patterns of data reuse. Section 4 describes two novel partitioning strategies in the dHPF compiler for improving performance: an implementation of multipartitioning for parallelizing line-sweep computations more effectively, and an out-of-core partitioning strategy for improving the performance of programs with very large data requirements. Section 6 compares the performance of compiler-generated parallelizations for serial versions of the NAS SP and BT application benchmarks with that of their hand-parallelized counterparts to illustrate the effectiveness of the analysis and code generation techniques described in Sections 3 and 4.

3. COMPUTATION PARTITIONING (CP) OPTIMIZATIONS

HPF compilers primarily use the *owner-computes* rule [13] to partition computations among a set of processors. This rule specifies that a computation is executed by the owner of the value being computed. This rule, as well as variants (e.g., in decHPF [19]) or more powerful rules (e.g., in SUIF [20]), can be expressed in terms of which processor(s) own a particular set of data elements.

In particular, for a statement enclosed in a loop nest with index vector $\underline{\mathtt{i}}$, and for some variable A, the CP ON_HOME $\mathtt{A}(\mathtt{f}(\underline{\mathtt{i}}))$, specifies that the dynamic instance of the statement in iteration $\underline{\mathtt{i}}$ will be executed by the processor(s) that own the array element(s) $\mathtt{A}(\mathtt{f}(\underline{\mathtt{i}}))$. This set of processors is uniquely specified by subscript vector $\mathtt{f}(\underline{\mathtt{i}})$ along with the distribution of array A at that point in the execution of the program. The owner-computes rule simply uses the left-hand-side reference of an assignment as the home reference in the CP. The rule in the SUIF compiler [20] is equivalent to using an arbitrary single reference as the home. SUIF, however, restricts all statements in a loop to the same computation partition, whereas the owner-computes rule does not.

The dHPF compiler supports a more general CP model [12]. A CP for a statement chosen by dHPF can be expressed as the owner of one *or more* arbitrary data references. Furthermore, each statement in a program may have its own independently chosen CP. The dHPF compiler implicitly represents the CP for a statement as a union of one or more ON_HOME terms:

$$CP : \cup_{j=1}^{j=n} \left\{ \mathtt{ON\_HOME}\ \mathtt{A_j}(\mathtt{f_j}(\underline{\mathtt{i}})) \right\}$$

This enables a very general class of partitionings, including many that are not possible in any previous compiler we are aware of (e.g., complex replicated partitionings generated by multiple ON_HOME terms, which have proved crucial in experiments with the NAS benchmarks). The compiler supports this more general class of partitionings by using a sophisticated and general (yet simple-to-implement) approach for communication analysis, optimization, and code generation based on symbolic integer sets [6], and using a powerful representation and enumeration algorithm for non-convex symbolic sets provided by the Omega library [21].

In this section, we describe three program optimization strategies used by dHPF that exploit the generality of its partitioning model by using complex computation partitionings to address some of the challenges described earlier. The analysis techniques and algorithms used to perform these optimizations are described in [12].

3.1. Parallelizing Computations using Partial Replication

As described in Section 2.1, the use of temporary arrays commonly induces sharing patterns that can be an obstacle to effective parallelization of loop nests. In Section 3.1.1, we describe how we exploit dHPF's general computation-partitioning model to partially replicate computation on elements of temporary arrays, thus reducing communication frequency and volume. Used judiciously, this approach can significantly improve parallel performance. In Section 3.1.2, we describe how a similar approach can be applied to improve parallel performance of computations on long-lived arrays as well.

3.1.1. Computations on Temporary Arrays

When each element of a temporary array used within a loop iteration is defined within that iteration before it is used and none of the array elements are used after the loop, such an array is said to be *privatizable* for that loop. For example, the array rhoq in Figure 1 is privatizable within the i loop. Identifying privatizable arrays is a challenging problem in general, but the NEW directive in HPF provides a mechanism for programmers to identify arrays as privatizable. (The use of this directive for the same example is shown in Figure 5.) Even when a compiler knows which arrays are privatizable, however, parallelizing the loop nest for a distributed-memory machine can still be a challenge.

```
   do 10 k = 1, grid_points(3)-2
CHPF$ INDEPENDENT NEW(cv,rhoq)
     do 10 i = 1, grid_points(1)-2
       do 20  j = 1-1, grid_points(2)-1
        ru1 = c3c4*rho_i(i,j,k)        ON_HOME lhs(i,j+1,k,2), lhs(i,j,k,3), lhs(i,j-1,k,4)
        cv(j) = vs(i,j,k)              ON_HOME lhs(i,j+1,k,2), lhs(i,j-1,k,4)
 20     rhoq(j) =                      ON_HOME lhs(i,j+1,k,2), lhs(i,j,k,3), lhs(i,j-1,k,4)
   *               dmax1(dy3+con43*ru1,dy5 + c1c5*ru1,dymax+ru1,dy1)
       do  30 j = 1, grid_points(2)-2
        lhs(i,j,k,1) =  0.0d0
        lhs(i,j,k,2) = -dtty2 * cv(j-1) - dtty1 * rhoq(j-1)
        lhs(i,j,k,3) =  1.0 + c2dtty1 * rhoq(j)
        lhs(i,j,k,4) =  dtty2 * cv(j+1) - dtty1 * rhoq(j+1)
30      lhs(i,j,k,5) =  0.0d0
10 continue
```

Figure 5. Computation partitioning for a loop nest from subroutine `lhsy` of the NAS SP computational fluid dynamics benchmark.

Consider again the loop nest from subroutine `lhsy` of benchmark SP, shown in Figure 1 In Section 2.1, we described three approaches to parallelizing the loop nest with privatizable arrays `cv` and `rhoq` based on conventional owner-computes partitionings. Each of these approaches has signficant drawbacks.

To avoid these drawbacks, the dHPF compiler exploits its general computation partitioning model to specify that each processor computes only those elements of the privatizable array that it will actually use [7]. When some array elements (such as the boundary elements of `cv` and `rhoq` in the example) are needed by multiple processors, the compiler partially replicates the computation of exactly those elements needed to achieve this goal. This is cost-effective in that it is the minimal amount of replication that completely avoids any communication of the privatizable array within the loop nest. The general CP model in dHPF is crucial for expressing the CPs required by this strategy. For a statement defining a privatizable variable, we assign a CP that is computed from the CPs of the statements in which the variable is used.

Figure 5 shows an annotated version of the loop nest of Figure 1. In each loop nest, for each assignment to a non-privatizable variable the compiler first chooses a CP using a global CP selection algorithm that attempts to minimize the communication cost for the loop nest. For the code shown in Figure 5, the algorithm chooses owner-computes CP for each of the five assignments to array `lhs`. The CPs for these assignments are represented by the boxed left-hand-side references. To compute the CPs for assignments to `cv` and `rhoq`, we then translate the CPs from the statements in which they are used. In Figure 5, the translations are indicated using dotted arrows for `rhoq` and dashed arrows for `cv`. For example, the use `cv(j-1)` has CP ON_HOME `lhs(i,j,k,2)`. This requires that the value of `cv(j)` be computed ON_HOME `lhs(i,j+1,k,2)`, so this CP term is included in the CP for the assignment to `cv`. The details of the translation of various subscripts in each CP term are described in [12].

The effect of the CP propagation phase is that each boundary value of `cv(j)` and `rhoq(j)` is computed on both of the processors adjoining the boundary, therefore avoiding communication for these arrays within the `i` loop. All non-boundary values of `cv` and `rhoq` are computed only on the processor that owns them. This strategy avoids costly communication inside the `i` loop, as well as needless replication of computation. Our experimental results show that without this optimization, the SP and BT benchmarks would be more than 16 times and 1.5 times slower respectively [12]. (This explains why the HPF versions of the NAS benchmarks developed by PGI [3] were extensively rewritten to eliminate such uses of privatizable arrays, despite the cost of significant increases in both code size and complexity.)

It is worth noting that the CP propagation strategy for `NEW` variables is insensitive to the data layouts of these variables. Regardless of the data layout directives specified for the `NEW` variables, CP propagation ensures that exactly those values needed on each processor are computed there. Any communication needed to obtain the operands for these replicated computations would depend on both data layouts and CP choices, but this is automatically handled by our integer-set framework which performs communication analysis, storage management, and code generation for arbitrary CPs and data layouts [6].

The example shown in Figure 2 and described previously in Section 2.1 can also be handled using dHPF's CP propagation for privatizable arrays. No suitable scope exists in which `fjac` and `njac` are privatizable. We remedy this by surrounding the pair of loop nests in the example with a single trip loop and annotate this new loop with a `NEW` directive for `fjac` and `njac`. As described earlier, the CPs for the assignments to `lhs` that use `fjac` and `njac` are propagated to the definitions of those privatizables, which causes the computation of the privatizables to be partially replicated along the boundaries of the data partitioning as necessary. Applying this optimization here avoids communicating values of `fjac` and `njac` and instead communicates only `u`, which is a factor of fifty smaller.

3.1.2. Computations on Long-lived Arrays

In the previous section, we described how the dHPF compiler partially replicates computations on privatizable arrays. In order to experiment with the same technique for arrays that are not privatizable, such as the long-lived arrays of Figure 6, we added a `LOCALIZE` [7] directive to dHPF. By marking a distributed array with `LOCALIZE` in an `INDEPENDENT` loop, a user asserts that all of the values of this distributed array will be defined within the loop before they are used within the loop, although they may also be used after the loop, i.e., they are not privatizable. The `LOCALIZE` directive signals the compiler that the computation of each element of the array should be done on the owner as well as on the processors that use the value of the element within the loop. Computing the values on the owner ensures that any later uses after the loop will be satisfied correctly. Any non-local operands required for computed the localized array will require communication to be inserted before the loop nest.

To eliminate the cost of boundary communication for the reciprocal variables in the example, we annotated them with the `LOCALIZE` directive and we added an outer one-trip loop in which to apply the directive. As for privatizable arrays, CPs for statements defining values of elements of a localized array are translated and propagated from statements that use these elements. For example, the statement computing `rho_i` gets the union of CP terms propagated and translated from the four uses of the array in the next loop nest, as well as the definition's owner-computes CP, `ON_HOME rho_i(i,j,k)`. As a result, the boundary communications of `rho_i` along the distributed directions can be eliminated.

```
CHPF$ INDEPENDENT LOCALIZE(rho_i, us, vs, ws, square, qs)
      do onetrip = 1, 1  ! enclosing one-trip loop
        do k = 0, N-1; do j = 0, N-1; do i = 0, N-1    Computation Partitionings Selected
          rho_i(i,j,k) = ...                ON_HOME rho_i(i,j,k), rhs(i,j+1,k,5), rhs(i,j-1,k,5) ...
          us(i,j,k) = ...                   ON_HOME us(i,j,k), rhs(i,j+1,k, 2:5), rhs(i,j-1,k, 2:5) ...
          vs(i,j,k) = ...                   ON_HOME vs(i,j,k), rhs(i,j+1,k,3), rhs(i,j-1,k,3) ...
          ws(i,j,k) = ...                   ON_HOME ws(i,j,k), rhs(i,j+1,k,4), rhs(i,j-1,k,4) ...
          square(i,j,k) = ...               ON_HOME square(i,j,k), rhs(i,j+1,k,2), rhs(i,j-1,k,2) ...
          qs(i,j,k) = ... square(i,j,k)     ON_HOME qs(i,j,k), rhs(i,j+1,k,5), rhs(i,j-1,k,5) ...
        ! x-direction ...
        do k = 1, N-2 ; do j = 1, N-2;  do i = 1, N-2     ! y-direction
          rhs(i,j,k,1) = ... u(i,j+1,k) ... u(i,j+1,k) ...
          rhs(i,j,k,2) = ... us(i,j+1,k) ... us(i,j-1,k) ...
          rhs(i,j,k,3) = ... square(i,j+1,k) ... square(i,j-1,k) ... vs(i,j+1,k) … vs(i,j-1,k) …
          rhs(i,j,k,4) = ... ws(i,j+1,k) ... ws(i,j-1,k) ...
          rhs(i,j,k,5) = … qs(i,j+1,k) ... qs(i,j-1,k)...
                       … rho_i(i,j+1,k) ... rho_i(i,j-1,k)...
          ...
        ! z-direction ...
```

Figure 6. Using `LOCALIZE` to partially replicate computation in subroutine `compute_rhs` from NAS SP.

Similarly we can avoid the communications for `us`, `vs`, `ws`, `square`, and `qs` in the `compute_rhs` with the partial replication of computation. `LOCALIZE` is also used in the subroutine `compute_rhs` from BT.

In the example, since there are no intervening definitions of `u` the elements of `u` used to compute the localized arrays are also used to compute the values of array `rhs` in subsequent loops. This coalescing of the communication needed for all these uses of array `u` eliminates the need to add more communication to compute the localized arrays, even after those computations are replicated. In this case, localization causes the compiler to eliminate communication of the reciprocals while not introducing any new communication. In general, however, replicating such computations can introduce new communication to obtain needed operands. Replication is beneficial when the cost of communication of the operands is less than the cost of transferring the values of the localized arrays. Making this decision should be straightforward to do with a compiler, although it is not currently implemented in dHPF.

3.2. Coping with Complex Patterns of Data Reuse

For loop nests with complex data dependences, such as the example of Figure 4, we have developed an algorithm to eliminate inner-loop communication without excessive loss of cache reuse. The algorithm is described in [12], and we illustrate its impact here using the above example. Figure 7 shows the CPs that are chosen by the algorithm for the example.

The algorithm combines intelligent CP selection with selective loop distribution. For each loop with loop-independent dependences, the algorithm first tries to assign identical CPs to all pairs of statements connected by such dependences, so that they always reference the same data on the same processor, thus avoiding communication. In this case, we say that the dependence is localized. Of course, it is not always possible to localize all loop-independent dependences. In such cases, the algorithm distributes the loop into the minimal number of loops required to break the remaining dependences.

For the loop nest in Figure 7, the loop-independent dependences from statement 1 to statements 2, 3, and 4 respectively will group all these four statements into one group, with their common CP set at the right hand side of statement 1. Similarly, the dependences from statement 2 to 5 and 8, from 3 to 6 and 9, from 4 to 7 and 10 will in turn restrict all the CP set for statement 5, 6, 7, 8, 9, and 10 to the same CP choice as statement group 1, 2, 3, and 4. Thus all the statements will have the same CP choice, as shown on the right-hand side of each statement. All the statements connected by loop-independent dependences are grouped successfully and no statement is marked for distribution. Note that the algorithm had to choose non-owner CPs for several statements in the loop. (It is important to note that the loop also requires fine-grain pipelined communication for several loop-carried dependences. This communication cannot be eliminated if the arrays are partitioned in the `j` dimension. The benefit of avoiding loop distribution is to eliminate the additional overhead that is caused by losing reuse for loop-independent dependences.)

Not all loops can have all their loop-independent dependences localized. In cases when a common CP choice does not exist for two groups of statements, the unaligned statements have to be marked to be distributed into different loops. In Figure 7, if statement 8 referenced `lhs(i,j+1,k,n+4)` instead of `lhs(i,j,k,n+4)`, there would be a loop-independent dependence edge from statement 6 to 8. This would requires that statement 6 be grouped with both statement 8 and statement 3. One of these requirements would fail, and thus we would have to distribute statement 6 and 8 or 6 and 3 into different loops. Significantly, this selective loop distribution algorithm will only distribute the loop nest into two new loops instead of into 10, as would result from a maximal distribution.

The loop distribution algorithm is applied for each loop nest by traversing the loop nest from the deepest loop outward. Thus all loop-independent communications are avoided when possible and the unavoidable ones are finally placed at the outermost loop-nest level when further distribution of the two end statements is illegal. Most loop nests in the NAS benchmarks do not need to be distributed at all after applying the CP selection step. For a small number of loop nests, the loop-distribution algorithm is able to place the communication at the outermost loop level by distributing the inside loop nest into only two new loop nests. The communication overhead due to loop-independent dependences is minimized while the original program loop structure is mostly preserved.

4. DATA PARTITIONING EXTENSIONS

This section describes two extensions to HPF that are implemented by the dHPF compiler to support scalable performance for line-sweep computations and programs manipulating very large arrays. Both of these can be viewed as applications of a more general overpartitioning strategy whereby each

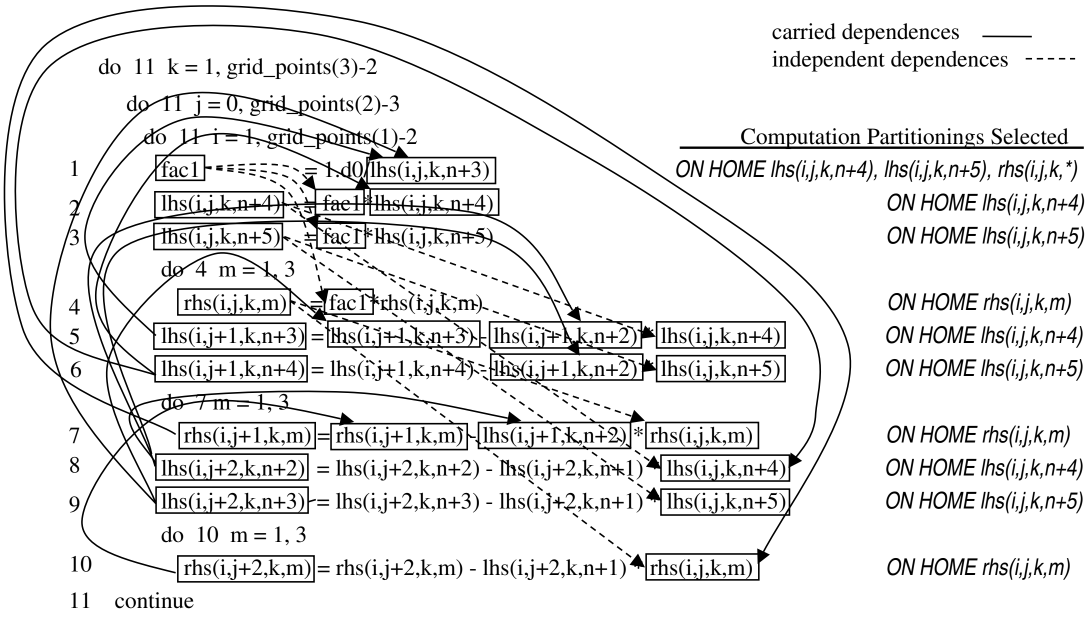

Figure 7. Communication-sensitive CP analysis for a loop nest from subroutine `y_solve` from SP.

processor manipulates more than one data tile. Another case in which overpartitionings apply is in handling `cyclic(k)` distributions.

The dHPF compiler's support for overpartitioning extends techniques originally developed for generating code for block partitionings of arbitrary size onto a symbolic number of processors. For the purposes of analysis, each tile in an overpartitioned array is treated as a block in a block-partitioned array of a higher order.

To support general overpartitioning, dHPF contains a general framework that controls how multiple *virtual processors* are mapped to each physical processor. The framework provides an interface that enables code generated for a single tile, using the virtual processor model, to be wrapped in a loop that iterates over all of the virtual tiles mapped to a processor. This interface ensures that the order in which tiles are processed, the tile schedule, is consistent with any data dependences that may be present. For data movement that has been vectorized out of a tile, the compiler provides mechanisms to coalesce the data-movement for all of the tiles owned by a physical processor, if desired.

In the next two sections, we describe how dHPF uses overpartitioning to support multipartitioned distributions and out-of-core computations on very large arrays.

4.1. Parallelizing Line Sweep Applications

Line sweeps are used to solve one-dimensional recurrences along each dimension of a multi-dimensional discretized domain. This computational method is the basis for Alternating Direction Implicit (ADI) integration — a widely used technique for solving partial differential equations such as the Navier-Stokes equation [14, 15, 16] — and is at the heart of a variety of other numerical methods and solution techniques [16].

Using line sweeps to solve recurrences along a dimension serializes computation of each line along that dimension. If a dimension with such recurrences is partitioned, it induces serialization between computations on different processors. Using standard block unipartitionings, in which each processor is assigned a single hyper-rectangular block of data, there are two classes of alternative partitionings. *Static block unipartitionings* involve partitioning some set of dimensions of the data domain, and assigning each processor one contiguous hyper-rectangular volume. To achieve significant parallelism for a line-sweep computation with this type of partitioning requires exploiting wavefront parallelism within each sweep. In wavefront computations, there is a tension between using small messages to maximize parallelism by minimizing the length of pipeline fill and drain phases, and using larger messages to minimize communication overhead in the computation's steady state when the pipeline is full. *Dynamic block unipartitionings* involve partitioning a single data dimension and performing line sweeps in all unpartitioned data dimensions locally, then transposing the data to localize the data along the previously partitioned dimension and performing the remaining sweep locally. While dynamic block unipartitionings achieve better efficiency during a (local) sweep over a single dimension compared to a (wavefront) sweep using static block unipartitionings, they require transposing *all* of the data to perform a complete set of sweeps, whereas static block unipartitionings communicate only data at partition boundaries.

To support better parallelization of line-sweep computations, a third sophisticated strategy for partitioning data and computation, known as *multipartitioning*, was developed [14, 15, 16]. Multipartitioning distributes arrays of two or more dimensions among a set of processors such that, for computations performing a directional 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 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. A study by van der Wijngaart [5] of implementation strategies for hand-coded parallelizations of ADI Integration found that 3D multipartitionings yield better performance than either static block unipartitionings or dynamic block unipartitionings.

4.1.1. Multipartitioning

Multipartitioning distributes multidimensional arrays such that for *any* directional sweep across the array, all processors are active in each step of the computation, there is perfect load balance, and only coarse-grain communication is needed. Figure 8 shows a 2D multipartitioning for 5 processors; a 2D array has been subdivided into 25 tiles, the number in each indicating its owning processor. For a line sweep in any possible direction (up, down, left, right), each processor can process one tile on each of the 5 steps in the sweep, with coarse-grain communication of boundary data required between steps.

Prepared using cpeauth.cls

x

y

0	4	3	2	1
1	0	4	3	2
2	1	0	4	3
3	2	1	0	4
4	3	2	1	0

Figure 8. 2D Multipartitioning on 5 processors.

Data partitionings that assign more than one data tile to each physical processor have been described previously in the literature [14, 15, 16]. "Diagonal"-style multipartitionings for two dimensions were described by Johnsson, Saad, and Shultz [15], and in three dimensions by Naik, Naik, and Nicoules [16]. Figure 8 shows a 2D multipartitioning for 5 processors, and Figure 9 shows a 3D multipartitioning for 16 processors.

These types of multipartitionings generalize to higher dimensions. Diagonal multipartitionings partition the data into $p^{\frac{d}{d-1}}$ tiles, where p is the number of processors and d is the number of partitioned array dimensions. Each physical processor is assigned $p^{\frac{1}{d-1}}$ tiles placed along diagonals through each of the partitioned dimensions. Diagonal multipartitionings apply only when the expression $p^{\frac{1}{d-1}}$ is an integer.

Recently, we have developed a generalization of multipartitioning to support d dimensional multipartitionings on p processors, where p is not restricted. This extension is another application of overpartitioning; each processor is assigned multiple tiles in each hyperplane [10].

4.1.2. *Implementation in* dHPF

Multipartitioning is not available in standard HPF, so we defined a new data distribution specifier called `MULTI`, which indicates that the corresponding dimension in the template should be multipartitioned. For example, the declarations

```
CHPF$    processors p(5)
CHPF$    template ta(0:9999,0:9999)
CHPF$    align a(i,j) with ta(i,j)
CHPF$    distribute ta(multi,multi) onto p
```

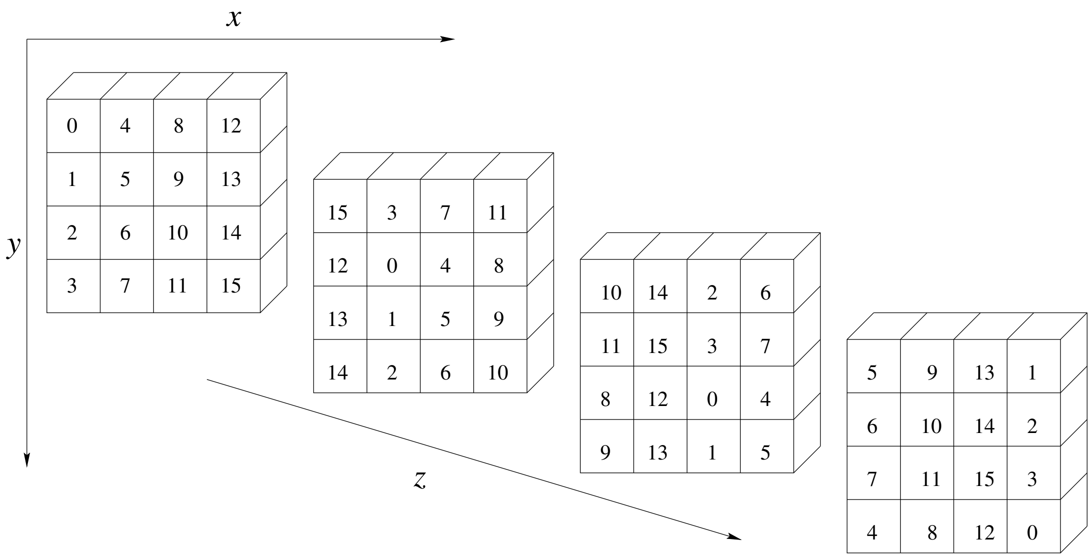

Figure 9. 3D Multipartitioning on 16 processors.

will generate a two-dimensional multipartitioning for the array `a`. The multipartitioned dimensions should be leftmost on the template; `ALIGN` directives can be used to implement different dimension orderings.

All multipartitioned dimensions are distributed among the number of processors indicated in the first dimension of the processors array. This allows for the possibility of combining multipartitioned and non-multipartitioned distributions on the same template. The dHPF compiler supports code generation for a symbolic number of processors for standard and multipartitioned distributions.

The integer-set analysis framework used by dHPF supports block partitionings of arbitrary size onto a symbolic number of processors. This framework serves as the basis for the dHPF implementation of multipartitioning. To support multipartitioned arrays, for the most part, dHPF treats each data tile of a multipartitioned array as a block, assigned to its own virtual processor, out of a block-partitioned array. To support a diagonal multipartitioning in d dimensions, dHPF generates SPMD code so that each physical processor manages $p^{\frac{1}{d-1}}$ virtual processor tiles.

Each data tile for a multipartitioned array is extended as appropriate with overlap areas [22] to facilitate access to non-local data received from neighboring 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 *local tile index*. All communication and computation performed for a tile are 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 to execute 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 handled similarly. Communication pinned inside a computational loop by data dependences is handled for a tile as if its virtual processor were a physical processor. Communication operations that are vectorized outside of all computational loops over data dimensions are enclosed in their own tile enumeration loop.

More details about multipartitioning and its implementation in dHPF can be found in [8, 23].

4.1.3. dHPF *Optimizations for Multipartitioning*

There are several key optimizations performed by the dHPF compiler that are necessary to generate high-performance multipartitioned code. From a single processor's perspective, a multipartitioned computation is organized as a series of computations on local tiles. To avoid unnecessary serialization between physical processors, each processor's local tiles should be scheduled in the proper order. To achieve good scalability, communication between processors must be as infrequent as possible.

Tile Scheduling & Tile Loop Placement Executing a multipartitioned computation corresponds to executing the same computation for each of a processor's tiles. Determining the order in which to iterate over these local tiles is not trivial in the presence of processor-crossing loop-carried dependences. Maintaining parallel execution and overall performance is a function of the order in which tile computations are performed on each processor.

This problem of selecting a tile enumeration order is unique to multipartitioned distributions. For block partitionings, the selection is trivial, since each processor owns only one block. Tile enumeration is also simple for cyclic and block-cyclic distributions; enumeration is independent for each partitioned dimension. Tile enumeration for multipartitioning is more complex because enumeration orders on any particular dimension determine the enumeration order for the other partitioned dimensions as well.

If a particular loop nest contains processor-crossing carried dependences flowing only in one dimension and direction of the partitioned array (i.e. line sweep phases), then selecting that dimension and direction for the tile iteration order will allow for maximum parallelism. If there are dependences flowing in both directions it is always possible to select computation partitionings (CPs) so that processor-crossing dependences flow in only one direction. Multidimensional dependences will necessarily restrict available parallelism, under multipartitioning [9].

Aggressive Communication Placement Processor-crossing loop-carried dependences can be preserved by placing communication for each dependence at the loop level on which it is carried. Loop-independent dependences can be preserved by placing communication at the loop level that is the least common ancestor of both source and sink.

For loop nests iterating over block-partitioned arrays, communication occurring at the block boundary can be hoisted out of the loop nest. This has the potential of reducing available parallelism while decreasing communication frequency. For multipartitioned distributions, if all communication is flowing along the same direction and dimension of the partitioned array, then it can be hoisted out of the loop nest completely without any loss in available parallelism.

In the dHPF compiler, we exploit this property to vectorize communication for processor-crossing dependences out of multipartitioned loops. This optimization enables the programmer to write the original HPF source loop nest in natural memory order without worrying about communication placement.

Communication Aggregation across Tiles A key property of multipartitioning 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, consider the tiles belonging to processor 1 in figure 9. All of their right neighbors belong to processor 5. Thus, if a tile need to shift data to the neighbor along a particular direction, all of that processor's tiles will also shift data in the same direction and the processor needs to send values to only one other physical processor.

This property must be exploited to obtain scalable performance in multipartitioned applications. If not, a separate message would be sent per tile for communication that has been vectorized outside of all loops over multipartitioned data. This would represent an overhead of $p^{\frac{1}{d-1}}$ more messages than necessary, inhibiting scalable performance for larger p.

4.2. Coping with Very Large Arrays

As described in Section 2, out-of-core (OOC) programs process such large amounts of data that they cannot all fit simultaneously within the available physical memory, with a consequent loss of performance. However, for many problems, data and program reorganization can lead to dramatic performance improvements, often achieving close to the expected in-core execution time.

The dHPF compiler can restructure such programs automatically for Out-Of-Core (OOC) execution. The dHPF compiler implements new directives [11], similar to HPF directives, for decomposing very large arrays into multiple tiles per processor, of which only a small subset are kept in-core at any time. For example, given the declaration `integer b(10000,10000)`, the directives

```
CSIO$    processors pio(100)
CSIO$    template tio(10000,10000)
CSIO$    align b(i,j) with tio(i,j)
CSIO$    distribute tio(*,block) onto pio
```

distribute the 100 million elements of `b` across 100 OOC tiles per processor. The dHPF compiler generates code that iterates over each OOC tile, reading it into memory, using it in computation, then saving it to disk. The compiler also generates code for managing any required communication between the OOC tiles.

The OOC distribution directives are orthogonal to the HPF directives, in that an array may be both distributed across real processors and also distributed across virtual 'OOC' processors. A limitation of the current implementation, however, is that each dimension can be distributed in at most one way.

OOC tiling was initially implemented by *ad hoc* code within the dHPF compiler. However, it became clear that there are significant similarities in the code required to implement OOC partitioning and multipartitioning. Each must subdivide the data assigned to each processor into multiple tiles, each must introduce code to iterate over the tiles allocated to each real processor, and each must manage communication between the tiles. The principle differences between OOC partitioning and multipartitioning are:

- OOC partitioning requires communication to be kept at the level of the loop that iterates over the tiles assigned to each processor.

- In the OOC case, communication associated with a particular tile must be scheduled to occur only when that tile is in memory. In the multipartitioning case, tiles are always in memory and thus there is much more opportunity for aggregating communication across tiles.
- Communication between tiles allocated to the same processor requires either I/O, retaining data in memory, or revisiting a tile. These are all very expensive.
- The traversal order of OOC tiles has a very large effect on performance.

Despite these differences, OOC partitioning and multipartitioning can both be implemented as specializations of a more general framework for supporting distributions that assign multiple tiles per processor. Currently, the dHPF compiler implements multipartitioning using such a framework. By unifying OOC handling with this framework as well, the implementation of the OOC distribution directives would be considerably simplified. Also, using a common framework for all forms of overpartitioning would provide greater flexibility for combining OOC and in-core distributions.

5. OTHER OPTIMIZATIONS

For compiler-generated parallelizations to achieve performance matching that of carefully hand-tuned code, it is clear that the compiler must get not just most, but *all* of the details perfect. For instance, although compiler-directed multipartitioned distributions lead to balanced computation, matching hand-coded performance also requires closely matching the communication frequency and volume, along with the scalar performance characteristics.

Therefore, in addition to the computation and data-partitioning optimizations described in previous sections, the dHPF compiler also performs a considerable number of additional optimizations to improve the efficiency of computation and communication. Although they are vital to achieving the performance results reported in the following section, we do not have space in this paper to describe in detail all of these strategies. In this section, we briefly mention some of these strategies and provide references, where available, to more detailed expositions.

- The compiler vectorizes communication for arbitrary regular communication patterns. Communication is vectorized out of any loop as long as doing so will not cause any loop-carried or loop-independent data dependence to be violated.
- The compiler can coalesce messages for arbitrary affine references to a data array. Any two communication events at a point in a program that are derived from different references to the same array will be coalesced if the data sets for the references overlap and both communication events involve the same communication partners. This optimization significantly reduces communication frequency.
- The compiler further reduces message frequency by aggregating communication events for affine references for disjoint sections of an array or different arrays if both communication events occur at the same place in the code and involve the same communication partners [23].
- The compiler-generated code and supporting runtime library use asynchronous communication primitives for latency and asynchrony tolerance [23].
- The compiler generates code which implements a simple array-padding scheme that eliminates most intra-array conflict misses [23], thereby improving cache performance.

- The compiler implements several context-sensitive guard reduction schemes to optimize guards and improve code efficiency. Information from the surrounding context determines whether a guard can be logically simplified by eliminating statically known conditions. These optimizations can significantly reduce guards and simplify control flow, specially for inner loops [24].

6. EXPERIMENTAL RESULTS

This section describes experimental results obtained using the dHPF compiler to generate optimized multipartitioned parallel code from serial versions of the NAS BT and SP application benchmarks.

In the parallelizations of these codes with dHPF, the optimizations described in the previous sections removed most of the performance bottlenecks. We briefly outline where these optimizations were applicable, compare the performance of the compiler-generated parallelizations to that of hand-coded MPI parallel implementations, and identify the causes of remaining performance differences.

Our experiments were performed on an SGI Origin 2000 (128 MIPS R10000 250MHz CPUs; 32KB (I)/32KB (D) L1, 4MB L2 (unified) for each CPU). Both the hand-coded and automatically parallelized applications used SGI's MPI library. No use was made of the shared memory capability of the SGI machines. All experiments with hand-coded MPI were performed in the default 32-bit mode. Experiments with dHPF-generated code were performed in 64-bit mode because dHPF's dynamic allocation of data arrays caused problems in 32-bit mode.

6.1. NAS BT

Effectively parallelizing the NPB2.3-serial version of BT with dHPF required coordinated application of a broad spectrum of analysis and code-generation techniques. The support for multipartitioning and new optimizations in dHPF improved both the performance and scalability of the generated code compared to earlier dHPF parallelizations [7] based on block partitionings. Both the parallel execution profile and the sequential performance of dHPF's compiler-multipartitioned code are nearly equivalent to the hand-coded version, which is a highly-tuned and very efficient parallel implementation. Figure 10 shows a 16-processor parallel execution trace for one steady-state iteration of our compiler-generated multipartitioned code for class A (64^3) problem size. Compared to the corresponding execution trace of the hand-coded multipartitioning shown in Figure 11, our dHPF-generated code achieves the same qualitative parallelization.

Tables I and II compare the speedups of the NASA Ames' hand-coded parallelization using multipartitioning with those of our dHPF-generated code. All speedups are relative to the execution time of the original sequential code. The columns labeled "% diff" show the difference, relative to speedup of the hand-coded version, between the speedup of our dHPF-generated version and the speedup of the hand-coded version.

For the class A problem size, dHPF's generated code was, at worst, about 15% slower than the hand-coded version. In several cases our code outperforms the hand-coded version.

Table I. Comparison of hand-coded and dHPF speedups for NAS BT (class A (64^3)).

# CPUs	hand-coded	dHPF	% diff.
1	1.06	1.13	-6.77
4	3.28	3.29	-0.11
9	7.73	7.15	7.49
16	14.21	13.19	7.21
25	21.08	18.41	12.67
36	29.78	28.61	3.94
49	39.73	33.93	14.60
64	48.13	54.21	-12.64

Table II. Comparison of hand-coded and dHPF speedups for NAS BT (class B (102^3)).

# CPUs	hand-coded	dHPF	% diff.
1	0.98	1.05	-7.76
4	3.37	3.28	2.76
9	4.91	8.03	-63.40
16	12.30	13.82	-12.35
25	19.09	21.55	-12.92
36	30.95	31.70	-2.44
49	52.82	41.60	21.24
64	66.04	54.21	17.91
81	82.28	64.37	21.77

For the class B problem size†, dHPF's generated code and the hand-coded version are also comparable, except on higher numbers of processors. This difference in performance comes from increased primary and secondary cache conflicts between data arrays and communication buffers in the dHPF generated code, compared to the hand-coded version. This is significant only on larger sets of processors because of the higher communication-to-computation ratio. In some of the cases, dHPF-generated multipartitionings achieved better performance than the hand-coded version.

The good parallel performance and scalability of the compiler-generated code comes from the application of the optimizations described in previous sections. Using non-owner computes computation partitionings to partially replicate computation along multipartitioned tile boundaries reduces communication volume dramatically. In BT's `compute_rhs` subroutine, partially replicating

†The poor measured speedup for the hand-coded 9-processor trial was likely caused by outside interference. An empty time allocation on our experimental platform kept us from re-running this measurement before publication.

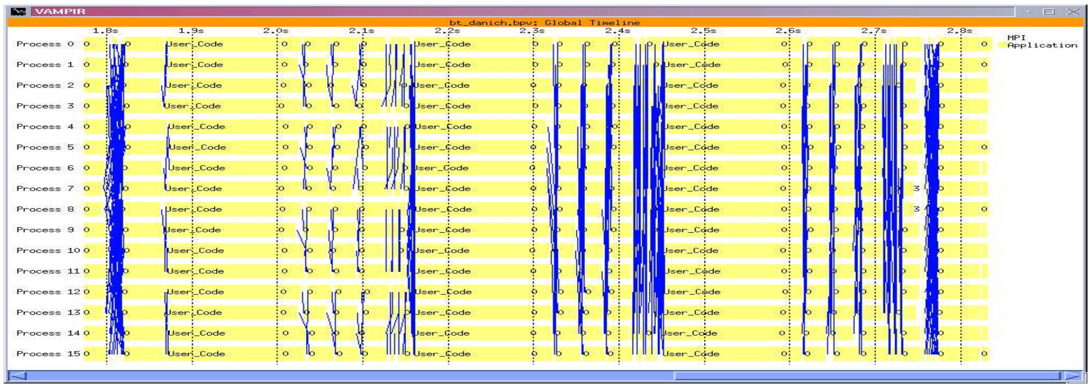

Figure 10. dHPF-generated NAS BT using 3D multipartitioning.

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 in this routine. (The redundant communication is eliminated using an additional optimization, namely communication coalescing [6], which was not described here.) Together, these optimizations 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 dimensions are `(5, 5, IMAX, JMAX, KMAX)`, reduced communication by a factor of five. 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, to replicate the computation along the partitioning boundaries.

The combined impact of these 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.

The number and frequency of the MPI messages generated by the compiler-generated BT code is very close to the corresponding pattern of MPI messages of the hand-coded version. The scalar performance of the two versions is also comparable, hence the small performance differential between the hand-coded version and the dHPF generated version. Figure 10 shows a 16-processor parallel execution trace for one steady-state iteration of the dHPF generated code for the BT class A sized benchmark. The corresponding hand-coded trace is shown in Figure 11. The traces show that the major communication phases are similar and occur in the same order. The principal differences are longer elapsed times in the hand-coded multipartitioning between sends and their corresponding receives, which was achieved by placing a section of the computation that does not depend on communication between the send and receive.

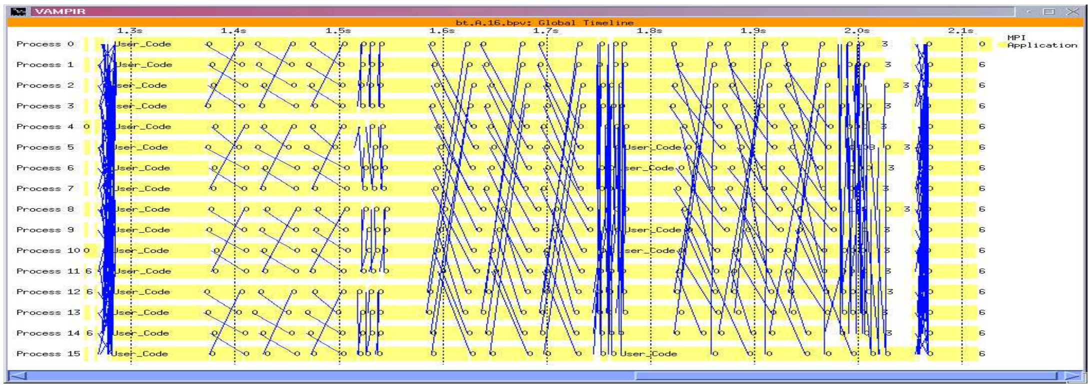

Figure 11. MPI hand-coded NAS BT using 3D multipartitioning.

6.2. NAS SP

Compiling the NAS SP benchmark for high performance presents a significant challenge. Generating efficient parallel code from a lightly modified serial version of this benchmark with a multipartitioning distribution requires the use of several advanced compilation strategies [7, 8], including non-owner-computes computation partitionings, complex patterns of computation replication to reduce communication for privatizable arrays, aggressive communication coalescing/aggregation, and a mix of complex computation partitionings.

Despite the fact that the dynamic communication patterns of our compiler-generated multipartitioned parallelization resemble those of the hand-coded parallelization, there is still a performance gap between our dHPF-generated code and the hand-coded MPI implementation. This gap comes in part 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 may schedule communication in each procedure even though the values might already be available as a result of prior communication. Interprocedural communication placement and interprocedural availability analysis would be needed to eliminate this additional communication automatically. The major contributors to the performance difference are primary and secondary cache conflicts caused by interference between data arrays and communication buffers, as described in Section 6.1.

Tables III and IV show speedup measurements for SP for the class A (64^3) and B (102^3) problem sizes, respectively. All speedups shown in the tables are relative to the performance of the sequential code for the respective NPB2.3-serial distribution.

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` increases the communication volume of our dHPF-generated code by 14.5% over that in the hand-coded parallelization. Although the additional

Prepared using cpeauth.cls

Table III. Comparison of hand-coded and dHPF speedups for NAS SP (class A).

# CPUs	hand-coded	dHPF	% diff.
1	1.01	0.96	5.50
4	4.21	3.54	15.87
9	11.60	9.22	20.54
16	16.21	16.94	-4.47
25	21.00	21.44	-2.08
36	30.69	34.37	-12.02
49	42.43	36.07	14.99
64	67.57	43.63	35.43

Table IV. Comparison of hand-coded and dHPF speedups for NAS SP (class B).

# CPUs	hand-coded	dHPF	% diff.
1	0.80	0.88	-9.05
4	2.86	2.60	9.23
9	7.74	6.98	9.78
16	13.01	13.97	-7.37
25	22.15	21.32	3.73
36	36.52	32.38	11.34
49	51.78	41.32	20.21
64	58.35	51.43	11.85
81	74.95	57.62	23.12

volume is modest, this unnecessary communication is responsible for the communication frequency of the dHPF-generated code being 74% higher than that of the hand-coded parallelization.

As 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 the boundary values between its definition and uses inside an inner loop. Although partially replicating computation of `cv` required communicating two additional planes of a three-dimensional multipartitioned array (`us`, `vs`, or `ws`) in each of these routines, these communications were fully vectorizable, whereas the communications of `cv` that we avoided were not.

In SP's `x_solve`, `y_solve`, and `z_solve` routines, the dHPF compiler chooses to generate extra fully vectorized communication if this will reduce communication pinned inside a loop nest. With the multipartitioning distribution, communication inside a loop nest does not have a high enough cost, and in the case of the sweep routines, a moderate increase in communication frequency should be traded for a significantly reduced total communication volume. By compiling these routines using an owner-computes scheme, we effectively get a reduced communication volume with no penalty due to the communication vectorization property of multipartitioned loops.

7. CONCLUSIONS AND FUTURE RESEARCH

This paper has described advanced optimization and code-generation strategies that allow one to write HPF source codes that can be compiled to efficient code without requiring the programmer to use extensive compiler-specific source code restructuring. Our research compiler, dHPF, implements all of these strategies.

The success of the dHPF compiler's optimizations is due principally to the formalization of data-parallel analysis and code generation as the manipulation of sets of integer tuples [6]. Sophisticated optimization techniques can be composed easily on top of this general framework. For example, it permits the compiler to generate code for a symbolic number of processors using a multipartitioned data distribution and to vectorize the communication resulting from the non-owner-computes computation partitionings that arise when computation is partially replicated at processor boundaries.

Another key to the dHPF compiler's success is a general overpartitioning framework made possible by the general integer-set equational framework. The overpartitioning framework provides a common infrastructure for such diverse paradigms as multipartitioning and out-of-core compilation. Out-of-core tiling addresses explicit data movement between memory and the disk system whereas multipartitioning addresses communication optimization, among other things. In both cases, the tiling induced by overpartitioning is combined with a static execution schedule that obeys dependence constraints while reducing the cost of data movement.

One of the weaknesses of HPF is the inflexibility of its built-in data distributions. Section 4 described two extensions to HPF's data distributions: multipartitioning and out-of-core partitioning. Both enable a broader class of efficient applications to be generated automatically by dHPF based on the use of appropriate data-distribution directives. An interesting topic for future research is the efficient implementation of arbitrary, user-defined data distributions. For example, adaptive distributions based on space-filling (for instance, Hilbert) curves [25] would enable many irregular applications to be implemented efficiently in HPF. A possible approach would be for the HPF compiler to generate code that uses a standard "data distribution" library interface, and for users to provide an implementation of that library interface for their new data distribution. A technique such as telescoping languages [26] could be used to make such an approach efficient. By extensively pre-compiling the user-supplied libraries, and by using performance-enhancing transformations specified by the library designer to guide the optimization process, the overhead of using such a packaged library might be significantly reduced.

With compiler-based support for multipartitioning, and the dHPF compiler's advanced optimization framework for reducing and aggregating communication, our experiments have shown that the performance gap between compiler-generated parallelizations for line-sweep computations and their hand-coded counterparts has been narrowed to just a few percent.

We believe that, with relatively little additional work, even these differences can be eliminated. Our experiments with the NAS SP benchmark, for example, show that the number of secondary cache misses by the dHPF compiler generated code is similar to that for the serial version on a single processor, but increases faster as the number of processors increases, leading, we believe, to the observed lower scalability of the dHPF generated code. These secondary cache misses arise from the generated buffer management code which is still not ideal. We believe that improving buffer management code will improve cache behavior and will eliminate the performance differential.

These results suggest that new compiler technologies may now make it possible for HPF to realize the original vision of its designers—that application developers should be able to produce efficient parallel applications from cleanly written Fortran programs by simply adding data-distribution directives, without having to tune the Fortran code to match the capabilities of the language processor. Had these technologies been available in HPF compilers early in the lifetime of the language, many application developers might have avoided the substantial effort required to convert their codes to MPI.

REFERENCES

1. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
2. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
3. The Portland Group. Hpf versions of the nas parallel application benchmarks, September 1998. Available from ftp://ftp.pgroup.com/pub/HPF/examples/npb.tar.gz.
4. D. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
5. 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.
6. Vikram Adve and John 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.
7. Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.
8. Daniel Chavarría-Miranda and John 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.
9. Daniel Chavarría-Miranda, John Mellor-Crummey, and Trushar Sarang. Data-parallel compiler support for multipartitioning. In *European Conference on Parallel Computing (Euro-Par)*, Manchester, United Kingdom, August 2001.
10. Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarría-Miranda. On efficient parallelization of line-sweep computations. In *9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, June 2001.
11. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.
12. Vikram Adve, Guohua Jin, John Mellor-Crummey, Ken Kennedy, and Qing Yi. Design and evaluation of a computation partitioning framework for data-parallel compilers. Technical Report CS-TR01-382, Dept. of Computer Science, Rice University, 2001.
13. 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.
14. 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, January 1988.
15. S. Lennart Johnsson, Youcef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 8(5):686–700, 1987.
16. N.H. 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.
17. Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler inserted releases to manage physical memory intelligently. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 31–44, October 2000.

18. Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1):5–20, January 1998.
19. J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal of Digital Equipment Corp.*, 7(3):5–23, Fall 1995.
20. 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.
21. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, April 1996.
22. M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.
23. Daniel Chavarría-Miranda, John Mellor-Crummey, and Trushar Tsarang. Data-parallel compiler support for multipartitioning. Technical Report CS-TR01-374, Dept. of Computer Science, Rice University, March 2001.
24. John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 26(5), 1998.
25. M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A common computational infrastructure for adaptive algorithms for PDE solutions. In *Proceedings of Supercomputing '97*, 1997.
26. K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*. Accepted for publication.