High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes

Vikram Adve
Department of Computer Science MS132
Rice University
6100 Main Street
Houston, TX 77005-1892.
adve@cs.rice.edu
www.cs.rice.edu/~adve

Guohua Jin
Department of Computer Science MS132
Rice University
6100 Main Street
Houston, TX 77005-1892.
jin@cs.rice.edu
www.cs.rice.edu/~jin

John Mellor-Crummey
Department of Computer Science MS132
Rice University
6100 Main Street
Houston, TX 77005-1892.
johnmc@cs.rice.edu
www.cs.rice.edu/~johnmc

Qing Yi
Department of Computer Science MS132
Rice University
6100 Main Street
Houston, TX 77005-1892.
qingyi@cs.rice.edu
www.cs.rice.edu/~qingyi

Abstract:

With current compilers for High Performance Fortran (HPF), substantial restructuring and hand-optimization may be required to obtain acceptable performance from an HPF port of an existing Fortran application. A key goal of the Rice dHPF compiler project is to develop optimization techniques that can provide consistently high performance for a broad spectrum of scientific applications with minimal restructuring of existing Fortran 77 or Fortran 90 applications. This paper presents four new optimization techniques we developed to support efficient

parallelization of codes with minimal restructuring. These optimizations include computation partition selection for loop nests that use privatizable arrays, along with partial replication of boundary computations to reduce communication overhead; communication-sensitive loop distribution to eliminate inner-loop communications; interprocedural selection of computation partitions; and data availability analysis to eliminate redundant communications. We studied the effectiveness of the dHPF compiler, which incorporates these optimizations, in parallelizing serial versions of the NAS SP and BT application benchmarks. We present experimental results comparing the performance of hand-written MPI code for the benchmarks against code generated from HPF using the dHPF compiler and the Portland Group's pghpf compiler. Using the compilation techniques described in this paper we achieve performance within 15% of hand-written MPI code on 25 processors for BT and within 33% for SP. Furthermore, these results are obtained with HPF versions of the benchmarks that were created with minimal restructuring of the serial code (modifying only approximately 5% of the code).

Keywords:
NAS benchmarks, HPF, parallelizing compiler

1. Introduction

Compilers and tools for High Performance Fortran (HPF) [8] are available for virtually every parallel machine, and have demonstrated good performance for some key benchmarks (within a factor of two of hand-coded message passing, and sometimes much closer). Nevertheless, current compilers may provide very widely varying performance for different data-parallel applications. Also, substantial restructuring and hand-optimization may be required to obtain acceptable performance from an HPF port of an existing Fortran application.

For example, recent performance results for HPF versions of the NAS parallel application benchmarks [4] show good or excellent performance and scalability, as compared with hand-coded message-passing (MPI) versions of the same benchmarks [3]. These results, however, were obtained by almost completely rewriting the benchmark source code as part of the conversion to HPF. The changes included partial conversion from Fortran 77 to Fortran 90, selective unrolling of loops and inlining of procedures, data copies to avoid wavefront (pipelined) communication, along with forward substitutions and loop realignments to avoid the use of privatizable arrays. The resulting codes were nearly twice the length of the original serial versions of the benchmarks.

The Rice dHPF compiler project aims to develop the compiler optimization techniques and programming tools required to achieve consistent performance across a broad class of data-parallel applications on a variety of scalable architectures. A key goal is 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. Two key features of the dHPF compiler are support for a flexible computation partitioning model that is more general than that used by previous research or commercial compilers, and the use of a powerful integer set framework to implement data-parallel program analysis and code generation [1, 2] These two features together have greatly facilitated both the implementation of core data-parallel optimizations as well as the development of novel optimizations.

Although the core data-parallel optimizations described in the literature are sufficient for high performance on kernels and small benchmarks, more realistic codes raise a number of new challenges

for parallelization in HPF. For example, carefully written scientific codes often make heavy use of privatizable array temporaries to hold intermediate values, and sometimes to minimize memory usage and maximize locality. While privatizable arrays can be identified in HPF using the NEW directive, sophisticated compiler support is required to effectively partition computation on privatizable arrays while avoiding excessive communication.

In this paper, we study the serial versions of the NAS application benchmarks to understand the challenges that arise in real-world applications and develop compiler techniques to address them. In the course of our work with these benchmarks, we developed several novel techniques that we have found to be important (and in some cases essential) for effectively parallelizing such codes. These include:

- computation partition selection for loop nests that use privatizable arrays, along with partial replication of boundary computations;
- commmunication-sensitive loop distribution to eliminate inner-loop communication without excessively sacrificing cache locality;
- interprocedural selection of computation partitions; and
- data availability analysis to avoid excess communication for values computed but not owned by the local processor.

Each of these techniques has been implemented in the dHPF compiler.

In this paper, we motivate and describe these techniques. Then we present preliminary experimental results to show that using these techniques, the dHPF compiler achieves two important goals. First, dHPF-generated code achieves execution times within 15% of hand-written MPI code on 25 processors for BT and within 33% for SP. Second, dHPF achieves this performance with only minimal restructuring of the serial codes in which approximately 5% of the source lines were modified to create the HPF version.[1]

2. Overview of the dHPF Compiler

This section provides a brief overview of the Rice dHPF compiler and its implementation of core data-parallel optimizations. More details are available in [1,2]. The compiler translates HPF programs into single-program-multiple-data node programs in Fortran 77 or Fortran 90 that use either MPI message-passing primitives or shared-memory communication. The focus of this paper is on code generation for message passing systems. In this section, we focus on describing several key aspect of the dHPF compiler, in particular, its computation partitioning (CP) model which is crucial to enable several of the optimizations described later; the CP selection algorithm used by the compiler (since several of the new optimizations work as extensions to this algorithm); the communication model which imposes some key assumptions on the compiler; and the integer set framework which supports the analysis and code generation for many key optimizations in the compiler.

The computation partitioning (CP) model in dHPF is a generalization of the *owner-computes rule*, a simple heuristic rule which specifies that a computation should be executed by the owner of the data element being computed (where the owner is determined by the data distribution). In dHPF, the CP for a statement can be specified as the owner of one *or more* arbitrary data references. For a statement in a loop nest with index vector **i**, the CP can be expressed in general as **ON_HOME A_1(i) union ... union A_n(i)**, where **A_1 ... A_n** are variables in the program with arbitrary HPF data distributions. The

owner-computes rule is then simply the special case were the CP is the owner of the single left-hand-side reference (for an assignment statement). However, the more general representation allows the compiler to use a much more arbitrary set of processors as the CP, including arbitrary processors other than the owner. This flexible representation plays a key role in several of the optimizations described in later sections.

In the absence of procedure calls, the dHPF compiler uses an explicit CP selection algorithm that selects CPs for one loop at a time within a procedure. For each loop, the algorithm constructs a number of candidate CP choices for each assignment statement within the loop, and then considers all combinations of choices for the different statements. For each combination of choices, it performs a simple approximate evaluation of the communication induced by that combination, estimates the communication cost, and then selects the combination that yields the minimum communication cost. To ensure correctness and to maximize parallelism in the presence of arbitrary contol-flow, the algorithm then propagates CPs to enclosing loops and control-flow (all the way out to subprogram entry points). Further details of local CP selection and propagation are not relevant here, and are beyond the scope of this paper.

The communication model in dHPF assumes that the owner of an array element (as determined by the data distribution) always contains the most up-to-date value of the element. Any processor that needs to access non-local data must obtain the data from the owner using appropriate communication and/or synchronization. Furthermore, any time a processor other than the owner computes a data element, it must communicate that value back to the owner before it can be made available to a third processor. The data availability optimization described in Section 4 is designed to avoid one of the key disadvantages of this communication model.

Most of the major data-parallel analyses and optimizations in the dHPF compiler are implemented using an abstract approach based on symbolic integer sets [1]. In this approach, the key data-parallel quantities the compiler operates upon are represented as sets of symbolic integer tuples. These include the iteration set for a loop, the data set for a reference or an array or a communication event, and the processor set for data distrbution or communication. Each optimization problem operating on these quantities is then expressed abstractly and very concisely as a sequence of integer set equations. This approach permits the formulation of optimizations to be simple and very concise, and yet much more general than the case-based formulations used in most data-paralel compilers.

We have used the CP model and integer set framework to implement several key optimizations in dHPF [1]. These include:

- aggressive static loop partitioning to minimize communication overhead;
- communication vectorization for arbitrary regular communication patterns;
- message coalescing for arbitrary affine references to an array;
- a powerful class of loop-splitting transformations that enable communication to be overlapped with communication, and reduce the overhead of accessing buffered non-local data;
- a combined compile-time/run-time algorithm to reduce explicit data copies for a message; and
- a control-flow simplification algorithm to improve scalar performance.

In addition, the compiler performs some important optimizations that do not use the integer set framework, including reduction recognition, dataflow-based communication placement, and overlap areas. Together, these optimizations have been shown to provide excellent parallel speedups for simple

kernels and benchmarks [1]. Our experiences with the NAS parallel benchmarks have indicated, however, that several additional techniques are needed to support constructs that commonly arise in larger, more realistic codes. These techniques are described in later sections of the paper. The following section gives a brief overview of the two NAS benchmarks studied in this paper.

3. NAS application benchmarks BT and SP

As described in a NASA Ames technical report, the NAS benchmarks BT and SP are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations [3]. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [3]. Both consist of an initialization phase followed by iterative computations over time steps. In each time step, boundary conditions are first calculated. Then the right hand sides of the equations are calculated. Next, banded systems are solved in three computationally-intensive bi-directional sweeps along x, y, and z directions. Finally, flow variables are updated. Values must be communicated among processors during the setup phase and the forward and backward sweeps along each dimension.

If the arrays are distributed in block fashion in one or more dimensions, the bi-directional sweeps along each distributed dimension introduce both serialization and intensive fine-grain communication. To retain a high degree of parallelism and enable coarse-grain communication along all dimensions, the NPB2.3b2 versions of BT and SP solve these systems using a skewed block distribution called multi-partitioning [3, 9]. With multi-partitioning, each processor handles several disjoint subblocks in the data domain. The subblocks are assigned to the processors so that there is an even distribution of work among the processors for each directional sweep, and that each processor has a subblock on which it can compute in each step of the sweep. Thus, the multi-partitioning scheme maintains good load balance and coarse grained communication.

The serial version of the NPB2.3 suite, NPB2.3-serial, is intended to be a starting point for the development of both shared memory and distributed memory versions of these codes for a variety of hardware platforms, for testing parallelization tools, and also as single processor benchmarks. The serial versions were derived from the MPI parallel versions of the benchmarks by eliminating the communication and having one processor compute the entire data domain. The serial versions of the code are nearly identical to their parallel counterparts, except for the lack of data domain partitioning and explicit communication. Therefore, the MPI version provides an excellent basis for evaluating the performance of compiler parallelization of the serial benchmarks. Both the hand-written and compiler-generated codes use non-blocking send/recv for communication.

4. Computation Partitions to Reduce Communication Overhead

In stencil computations, frequent communication of boundary values can often be avoided by replicating copies of these values on each processor that needs them. This approach can reduce communication costs and, in some cases, reduce load imbalance as well. In the TOPAZ tool, Sawdey and O'Keefe explored replicating computation of boundary values for single-program-multiple-data (SPMD) Fortran codes [11]. TOPAZ reads a SPMD Fortran program and, for marked regions of code, it determines what width of overlap areas is needed to hold off-processor boundary values that will be accessed. Next the tool rewrites code within the marked region by expanding loop bounds to replicate computation of

boundary values into each processor's overlap areas. Forcing each processor to compute not only its local section of data but also the data in its overlap area eliminates the need for much of the communication and synchronization within marked regions of SPMD stencil codes. Here we describe two different aspects of a computation partitioning strategy for an HPF compiler that make similar use of partially replicated computation as necessary to reduce communication.

4.1 Parallelizing Computations that use Privatizable 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 the same or subsequent iterations. Typical uses of a temporary array are to save a reference copy of values before they will be overwritten, or to store partial results of a computation that will be used later.

Indiscriminate use of temporary arrays can be an obstacle to effective parallelization of a loop nest because they introduce dependences between different loop iterations. However, in the special case when each element of a temporary array used within a loop iteration is defined within that iteration before its use, and none of the elements defined in the loop are used after the loop, the iterations of the loop nest are fully parallelizable. Such an array is said to be *privatizable* on the loop. The High Performance Fortran NEW directive provides a convenient syntactic mechanism for programmers to signal a compiler that an array is privatizable.

Even once a compiler knows which arrays are privatizable within a loop nest, it can be challenging to effectively parallelize the loop nest for a distributed-memory machine. For example, consider the following loop nest from subroutine lhsy of the NAS 2.3-serial SP benchmark, with HPF directives added.

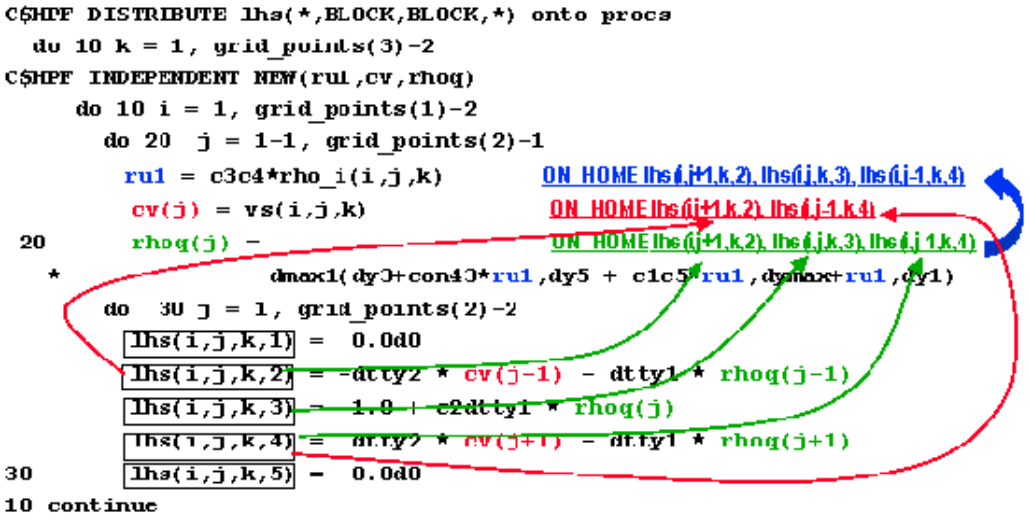


Figure 4.1: Computation partitioning for a loop nest from subroutine lhsy from SP

In this example, the NEW directive specifies that the **cv** and **rhoq** arrays are privatizable within the **i** loop. The difficulty in this case arises in trying to exploit the potential parallelism of the **j** loop. When computing the privatizable arrays **cv** and **rhoq**, there are two obvious alternatives, both of which are

unsatisfactory here. If a complete copy of the privatizable array is maintained on each processor, each processor will needlessly compute all the array elements even though only about **1/P** of the values will be used by each processor within the second **j** loop nest. On the other hand, if the privatizable array is partitioned among processors and each processor computes only the elements it owns, then each processor will have to communicate the boundary values of **rhoq** and **cv** to its neighboring processors for use in the second **j** loop. This would require a large number of small messages between processors, which would also have very high overhead.

In the dHPF compiler, we address these problems by having each processor compute *only* those elements of the privatizable array that it will actually use. 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 to achieve this goal. This is cost-effectve in that it is the minimal amount of replication that completely avoids any communication of the privatizable array in the inner loop. 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 each of the statements in which the variable is used. This is done as follows.

For each loop nest, the compiler first chooses CPs for all assignments to non-privatizable variables using a global CP selection algorithm that attempts to minimize communication cost for the loop nest. For the code shown in Figure 4.1, the algorithm chooses owner-computes CPs for each of the five assignments to array **lhs**. The CPs for these assignments are represented by the boxed left-hand-side references. We then translate these CPs and apply the translated CPs to the statements that define the privatizable arrays, **cv** and **rhoq**. In the figure, the arrows show the relationship between definitions and uses.

There are three steps to translating a CP from a use to a definition. First, where possible, we establish a one-to-one linear mapping from subscripts of the use to corresponding subscripts of the definition. For the case of the use **cv(j-1)**, this corresponds to the mapping **[j]**$^{\mathbf{def}}$ **-> [j-1]**$^{\mathbf{use}}$. (Note that the **j** on the left and **j** in the right refer to two different induction variables that just happen to have the same name.) If it is not possible to establish a 1-1 mapping for a particular subscript, or the mapping function is non-linear, this step is simply skipped. Next, we apply the inverse of this mapping to the subscripts in the **ON_HOME** references in the CP of the use. This translates **ON_HOME lhs(i,j,k,2)** to **ON_HOME lhs(i,j+1,k,2).** Finally, any untranslated subscripts that remain in the CP from the use are *vectorized* through any loops surrounding the use that do not also enclose the definition. (There are no such subscripts in this example.) Vectorization transforms a subscript that is a function of a loop induction variable into a range obtained by applying the subscript mapping to the loop range. The statement defining the privatizable array gets the union of the CPs translated from each use in this fashion. In the figure, a translation is applied from the use CP to a definition CP along each of the red and green arrows. For a privatizable scalar definition, the process is simpler. There is no subscript translation to perform; CPs from each use are simply vectorized out through all loops not in common with the definition. The blue arrow represents a trivial vectorization (a simple copy) of the CP from the uses of the privatizable scalar **ru1** to its definition since the use CPs are on statements in the same loop.

The effect of this CP propagation phase is that each boundary value of **cv(j)** and **rhoq(j)** is computed on both processors at either side of 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. Through this example, we have illustrated how our strategy avoids both costly communication inside the **i** loop, as well as needless replication of computation.

It is worth noting that this CP propagation strategy for NEW variables is insensitive to the data layout of these variables. Regardless of what data layout directives for the NEW variables may be specified, CP propagation ensures that *all and only* the values that are needed on each processor are computed there.

4.2 Partial Replication of Computation

To support efficient stencil computations on distributed arrays, we developed support in dHPF for a new directive we call LOCALIZE. By marking a variable with LOCALIZE in an INDEPENDENT loop, the user asserts that all of the values of the distributed array that has been marked will be defined within the loop before they are used within the loop. Furthermore, the LOCALIZE directive signals the compiler that the assignment of any boundary values of the marked arrays that are needed by neighboring processors within the loop should be replicated onto those processors *in addition* to being computed by the owner. Unlike the NEW directive which requires that values assigned to a marked variable within the loop are not live outside the loop, variables marked with LOCALIZE for a loop may be live after the loop terminates. Marking a variable with LOCALIZE for a loop has the effect of ensuring that no communication of the marked variables will occur either during the loop or as part of the loop's finalization code. Instead, additional communication of values will occur before the loop to enable boundary value computations for the marked variables to be replicated as required to ensure that each use of a marked variable within the loop will have a local copy of the value available. To explain how the dHPF compiler uses its general CP model to implement such partial replication of computation, consider the following loop nest from subroutine compute_rhs of the NAS 2.3-serial BT benchmark, with HPF directives added.

```
C$HPF DISTRIBUTE (BLOCK, BLOCK, BLOCK) onto procs :: rho_i, us, vs, ws,
square, qs
C$HPF INDEPENDENT, LOCALIZE(rho_i, us, vs, ws, square, qs)
      do onetripL = 1, 1
         // reciprocal computation
         do k = 0,N-1
            do j = 0,N-1
               do i = 0,N-1
                  rho_i(i,j,k) = ...
                  us(i,j,k) = ...
                  vs(i,j,k) = ...
                  ws(i,j,k) = ...
                   square(i,j,k) = ...
                   qs(i,j,k) = ...
                   ...
          // xi-direction
          do k = 1,N-2
             do j = 1,N-2
                do i = 1,N-2
                   rhs(2,i,j,k) = ... square(i+1,j,k) ... square(i-1,j,k) ...
                   rhs(3,i,j,k) = ... vs(i+1,j,k) ... vs(i-1,j,k) ...
                   rhs(4,i,j,k) = ... ws(i+1,j,k) ... ws(i-1,j,k) ...
                   rhs(5,i,j,k) = ... qs(i+1,j,k) ... qs(i-1,j,k)
                                    ... rho_i(i+1,j,k) ... rho_i(i-1,j,k)...
```

```
        ...
    // eta-direction
    ...
    // zeta-direction
    ...
enddo
```

Figure 4.2: Using LOCALIZE to partially replicate computation in subroutine compute_rhs from BT

In compute_rhs, **rhs** is evaluated along each of the **xi**, **eta**, and **zeta** directions. Along each direction, flux differences are computed and then adjusted by adding fourth-order dissipation terms along that direction. To reduce number of operations (especially floating-point divisions) reciprocals are computed once, stored, and then used repeatedly as a multiplier at the expense of storage for the reciprocal variables **rho_i**, **us**, **vs**, **ws**, **square** , and **qs**. Without partial replication of computation, the best CP choice for each statement that defines a reciprocal would be **ON_HOME var(i,j,k)** (where **var** stands for one of the reciprocal variables), and the statements that add dissipations would be **ON_HOME rhs(m,i,j,k)** ($1 < m < 6$). However, this CP choice causes each subsequent reference to the reciprocal variables **var(i-1,j,k)** and **var(i+1,j,k)** in the **rhs** computation along the **xi** direction to become non-local. Similar references along other two directions become non-local as well. In this case, the boundary data of all these arrays would need to be communicated, which can be costly in distributed memory systems.

To reduce the cost of boundary communication for the reciprocal variables, we added an outer one trip loop to define a scope in which to LOCALIZE the reciprocal variables. In a fashion similar to how we calculate CPs for statements defining privatizable arrays, CPs for statements that define elements of arrays marked LOCALIZE are translated and propagated from statements that use these elements later. Consider the reciprocal variable **rho_i** in Figure 4.2. To calculate the CP for the statement defining elements of **rho_i**, we translate the **ON_HOME** CP at each of the use sites and propagate it back to the definition site. For the **xi** direction flux computation we translate **ON_HOME rhs(5,i,j,k)** to **ON_HOME rhs(5,i+1,j,k)** from the use **rho_i(i+1,j,k)**, and **ON_HOME rhs(5,i-1,j,k)** from the use **rho_i(i-1,j,k)**. The same translation process needs to be performed for uses in the **eta** and **zeta** directions. The CP of the statement that defines **rho_i** finally is set to the union of all the propagated CPs as well as the definition's owner-computes CP **ON_HOME rho_i(i,j,k)**. Thus, the CP for the definition of **rho_i** becomes the union of **ON_HOME rhs(5,i+1,j,k)**, **ON_HOME rhs(5,i-1,j,k)**, **ON_HOME rhs(5,i,j+1,k)**, **ON_HOME rhs(5,i,j-1,k)**, **ON_HOME rhs(5,i,j,k+1)**, **ON_HOME rhs(5,i,j,k-1)**, and **ON_HOME rho_i(i,j,k)**. By partially replicating computation, computation of boundary data is executed not only on the processor which owns the data, but also on processors that need the data. As a result, the boundary communications of **rho_i** along all three directions can be eliminated. Similarly we can avoid the communication for **us**, **vs**, **ws**, **square**, and **qs**in the compute_rhs with the partial replication of computation. We applied this same technique to compute_rhs in SP.

Though partial replication of computation itself might introduce additional communication for the statements at the definition sites as an effect of the CP selection, it is beneficial when the number of messages and volume of the data to partially replicate the computation is smaller than transferring the values of the arrays marked LOCALIZE.

5. Commmunication-Sensitive Loop Distribution

Large loop nests in scientific codes often have substantial intra-loop reuse of array values, both for privatizable and non-privatizable arrays. This reuse appears as loop-independent data dependences within each loop iteration. If two statement instances connected by such a dependence are assigned to different processors, communication would be induced in the inner loop at considerable cost. Loop distribution is the most common technique used to avoid such inner loop communication, but if used too aggressively, it increases cache misses by disrupting temporal reuse.

We have developed an algorithm to eliminate such inner-loop communication, combining 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 loop-independent dependences, so that they always reference the same data on the same processor, thus avoiding communication. In this case we say the dependence is localized. Of course, it's not always possible to localize all loop-independent dependences. In such cases, the algorithm distributes the loop into a minimal number of loops to break the remaining dependences.

Figure 5.1 shows a loop nest from subroutine y_solve of NAS benchmark SP. In the figure, the dependences are shown as arcs connecting references within statements. Let's assume that both arrays **`lhs`** and **`rhs`** are block partitioned on the **`j`** and **`k`** dimension.

Our CP selection / loop distribution algorithm for loop-independent dependences works with the basic CP selection strategy of the compiler described in Section 2, by trying to constrain the set of choices selected to ensure statements connected by loop independent dependences have at least one common CP choice available. When applied to a loop nest, it first annotates each statement inside the loop with the set of CP choices selected for that statement by the basic CP selection algorithm, with a CP choice correspond to each different partitioned array reference in the statement. Because statements 1, 2, 3 and 10 only have one partitioned array reference, their annotated CP choice set will only contain one item, as shown on the right hand side of the statements. All the other statements have two different partitioned array references (different array references with the same data partition will be considered identical). For example, statement 5 will be annotated with CP set **`{ON_HOME lhs(i,j+1,k,n+3), ON_HOME lhs(i,j,k,n+4)}`**.

After each statement is annotated with a CP set, the algorithm begins to group the statements together by restricting their CP sets to their common CP choices. Initially each statement belongs to a separate group consisting of only itself. For each loop-independent dependence inside the loop, the algorithm tries to group the two end statements together if they do not already belong to the same group. Their common CP choices is computed by intersecting their currently associated group CP choices. If the set of common CP choices is not empty, each group's CP choice set is restricted to the common choices, and these two groups of statements are unioned together into one group. If no common CP choices are available, the two end statements are marked to be later distributed into different loops. This algorithm for restricting CP choices can be implemented using the classic union-find algorithm, thus taking close to linear time in the number of loop-independent dependence edges.

In the loop nest shown in Figure 5.1, the loop-independent dependences from statement 1 to statements 2, 3, and 10 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 10 to 20 and 30 will in turn restrict all the CP set for statement 5, 6, 10, 8, 9 and 30 to the same CP choice as statement group 1, 2, 3, and 30. Thus all the statements will have the identical CP choice, as shown on right hand side of each statement. All the statements connected by

loop-independent dependences are grouped successfully and no statement is marked for distribution.

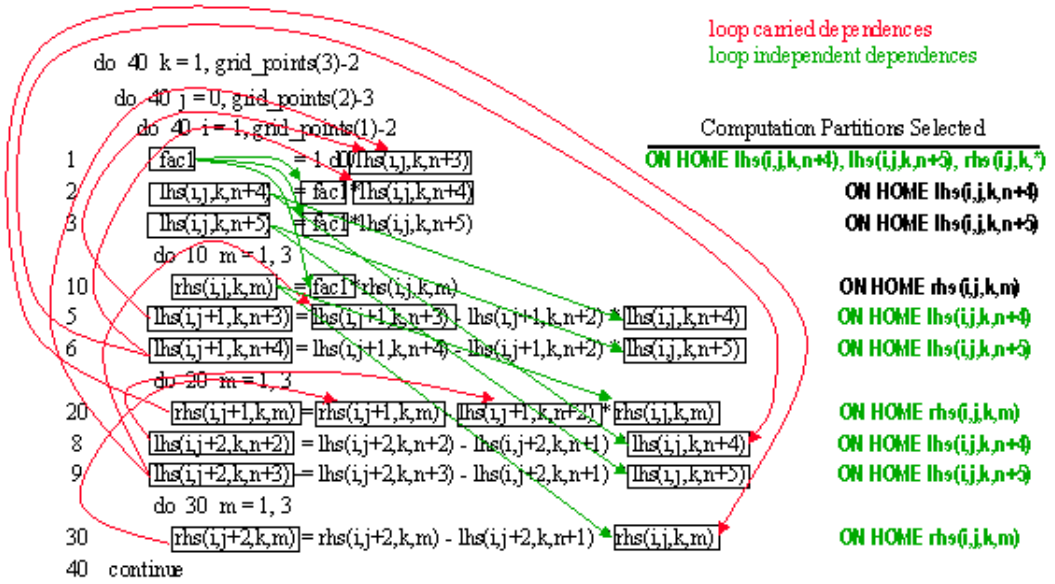


Figure 5.1: Communication-sensitive CP analysis for a loop nest from subroutine y_solve from SP.

Not all loops can have all its loop-independent dependences localized as can the loop in Figure 5.1. 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 5.1, if statement 8 references **`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. Since we would both need to group statement 6 with statement 8 and statement 3, one of the efforts will fail and thus we would have to distribute statement 6 and 8 or 6 and 3 into different loops.

To distribute the loop, like all the other loop distribution algorithms, a graph is constructed with a node for each statement and an edge for each dependence inside or carried by the loop; all the SCCs (strongly connected regions) of the graph are identified. At this point, a normal loop distribution algorithm would distribute each SCC group of statements into a separate new loop; our algorithm only selectively distributes these SCCs. For each pair of statements marked to be distributed, if they belong to different SCCs, The two SCCs are marked to be distributed into separate loops. A simplified version of the classic loop fusion algorithm is then applied to fuse together the SCCs not need to be distributed. The least number of new SCC groups are produced and each new SCC group is transformed into a new loop. This step takes time linear in the number of statements. In Figure 5.1, if statement 6 and 8 are marked to be distributed, the **`i`** loop will only be distributed into two new loops, instead of 10 new loops which would result from a maximum 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 distributing 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.

6. Interprocedural Selection of Computation Partitionings

It is common for large data-parallel codes to use function calls within parallel loops to implement pointwise or columnwise computations (or in general computations on some subset of the dimensions of the data domain). For example, function calls are used in key parallel loops of many of the NAS benchmarks including SP, BT and LU to carry out computations over one or two non-distributed dimensions. While such loops can be designated as parallel using the HPF INDEPENDENT directive, choosing how to partition the computation for such loops effectively can be difficult.

The local CP selection algorithm used by the dHPF compiler in the absence of procedure calls was described in Section 2. The compiler uses an extension of this local CP selection algorithm to support interprocedural parallelization. The algorithm proceeds bottom-up on the procedures in the call-graph. First, for leaf procedures in the call-graph, the algorithm uses the local CP selection algorithm unchanged. This step will cause a CP to be assigned to each entry point of each leaf procedure. In non-leaf procedures, for each statement containing a function call, the algorithm restricts the candidate CP choices to a single choice corresponding to the CP assigned to the entry point of the callee (available since the algorithm proceeds bottom-up on the call-graph). Otherwise, the local CP selection algorithm proceeds unchanged for the procedure, enumerating CP choices for each statement within a loop and choosing the least-cost combination. This one-pass, bottom-up algorithm naturally ensures that the data sub-domain parallelism is realized effectively.

The major challenge that arises in translating a callee's CP to a call site is that the CP may be defined in terms of local variables in the callee that may not be visible in the caller. To avoid this problem, we take advantage of the availability of templates in HPF: we first translate the CP in terms of template elements and then translate that interprocedurally. E.g., if array element **`A(i,j)`** is aligned with template element **`T(i-1,j)`** then the CP **`ON_HOME A(i,j)`** is equivalent to **`ON_HOME T(i-1,j)`**. In translating this to the call sites, if no equivalent template **`T`** is available in one of the callers, we simply synthesize a template with the same shape and distribution as **`T`** when performing CP selection for that caller. Note that any other necessary translations such as formal argument to actual name or value must also be performed for the expressions in the **`ON_HOME`** terms.

```
CHPF$ DISTRIBUTE rhs(*,*,BLOCK,BLOCK) onto procs
CHPF$ DISTRIBUTE lhs(*,*,*,*,BLOCK,BLOCK) onto procs
 do k=1,grid_points(3)-2
   do j=1,jsize-1
CHPF$ INDEPENDENT, NEW(temp_rhs, temp_lhs)
     do i=1,grid_points(1)-2
       temp_rhs(:) = rhs(:,i,j-1,k)                ON HOME rhs(1,i,j,k)
       call matvec_sub(i,j,k,lhs(1,1,aa,i,j,k), temp_rhs,
                       rhs(1,i,j,k))
       temp_lhs(:,:) = lhs(:,:,cc,i,j-1,k)         ON HOME lhs(1,1,bb,i,j,k)
       call matmul_sub(i,j,k,lhs(1,1,aa,i,j,k), temp_lhs,
                       lhs(1,1,bb,i,j,k))
       call binvcrhs(i,j,k,lhs(1,1,bb,i,j,k),
                     lhs(1,1,cc,i,j,k), rhs(1,i,j,k))
     enddo
   enddo
 enddo
```

Figure 6.1: Interprocedural selection of computation partitionings

To illustrate the overall algorithm, consider the code fragment from NAS BT in Figure 6.1. In the figure, the assignments to **temp_rhs** and **temp_lhs** did not exist in the original code but were added for the reasons explained below. The three called subroutines matvec_sub, matmul_sub and binvcrhs are all leaf routines. During CP selection for subroutine matvec_sub (for example), the compiler chooses a CP for the entire subroutine that is equivalent (after translation through templates) to **ON_HOME rhs(1,i,j,k)**. This is exactly as if the owner-computes rule were applied to the entire subroutine body, since **rhs** is the output parameter of the call. The call statement is therefore assigned exactly this CP. Similarly, the calls to matmul_sub and binvcrhs are assigned to execute **ON_HOME lhs(1,1,bb,i,j,k)** and **ON_HOME rhs(1,i,j,k)** respectively.

One point of subtlety in the example is that without a Fortran 90 array section subscript, an interface block, or interprocedural array section analysis, the set of values to be used by the called routine are unknown. We put in the explicit copies to **temp_lhs** and **temp_rhs** in the caller, as shown in the figure as a stopgap solution to make the communication for the callee's data usage explicit. Given these explicit copies, the compiler is able to hoist this communication out of one additional loop in the caller and realize the available parallelism of the enclosing loops. Given a description of the data the callee will access with one of these other mechanisms, the non-local values of **lhs** and **rhs** could simply be fetched into overlap regions and the copy of local values into the temporary could be avoided.

7. Data Availability Analysis

The dHPF compiler frequently selects CPs in which an array element is computed by a processor other than the owner. According to the communication model assumed in dHPF (see Section 2), these values are sent back to the owner before being available for any further communication. Frequently, however, these non-local values may later be needed by the same processor that has computed them. Ordinarily, such accesses would be marked non-local and generate communication to fetch the data from the owner. If we can determine (at compile-time) that such a non-local read will be preceded by a non-local write

on the same processor, i.e., that the value is actually available locally, then we can eliminate the communication for the non-local read reference. Determining this precedence is a complex problem in general, requiring both dataflow analysis and symbolic array section kill analysis [6]. Nevertheless the benefits can be substantial. For example, the selective loop distribution algorithm causes such CPs to be selected for pipelined computations in SP, and the extra non-local read communication flows in the opposite direction to the pipeline, causing the pipeline performance to be quite poor. Eliminating this communication proved essential for obtaining an efficient pipeline.

In dHPF, we use sophisticated array section analysis to compute the local availability of non-local data. This algorithm leverages our integer-set based communication analysis framework. For a given non-local read communication, we consider each non-local read reference causing this communication. For each such reference, we use dependence information to compute the last write reference that produces values consumed by that read. (Even if there are multiple write references reaching a read, we conservatively only consider the last write because we currently cannot take kill information into account, which is required for correctly handling multiple writes.) The communication analysis algorithm in dHPF already computes the set of non-local data accessed by each non-local reference (read or write), on the representative processor myid [1]. Therefore, all we have to do is to check if the non-local data accessed by the read is a subset of the non-local data produced by the write. In that case, the data is locally available and the communication can be eliminated. This algorithm directly eliminates about half the communication that would otherwise arise in the main pipelined computations of SP.

To illustrate the algorithm, consider again the source code loop nest from NAS SP shown in Figure 5.1. The assignments to **`lhs(i,j+1,k,n+3)`** and **`lhs(i,j+2,k,n+3)`** are both non-local writes because they both get the CP **`ON_HOME lhs(i,j,k,n+4)`**. In the former statement, the read reference to **`lhs(i,j+1,k,n+3)`** is also non-local for the same reason. The value accessed by this read is exactly the value written by the non-local write to **`lhs(i,j+2,k,n+3)`** in the previous iteration of the **`j`** loop (which is the last preceding write), and it would have been written by the same processor (as long as the block size of the BLOCK distribution is greater than 1). Dependence analysis proves that this is the only preceding write within the loop nest (note that it corresponds to the deepest dependence). Let M_j denote the processor id of the representative processor in the **`J`** dimension, B_j denote the block size of the block distribution in this dimension, and G_1, G_2, G_3 denote the values of **`grid_points(1)`**, **`grid_points(2)`**, and **`grid_points(3)`** respectively. Then the non-local data acccessed by the read reference is given by:

$$nonLocalReadData := [1 : G_1-2,\ M_j*B_j+B_j+1,\ 1 : G_1-2,\ n+3]$$

while the non-local data computed by the preceding write reference is:

$$nonLocalWriteData := [1 : G_1-2,\ M_j*B_j+B_j+1 : M_j*B_j+B_j+2,\ 1 : G_1-2,\ n+3].$$

Since the former is a subset of the latter, the communication for the read can be entirely eliminated. This read communication would have flowed in the direction of decreasing processor ids in this processor dimension whereas the pipelined communication for the two non-local writes flows in the opposite direction. Thus, this read communication would completely disrupt the pipeline, and eliminating it is crucial to obtaining an efficient pipeline. Note that the read reference to **`lhs(i,j+2,k,n+3)`** in the second of the two assignments mentioned above also causes communication which cannot be eliminated, but this communication occurs before the loop nest begins and is therefore not disruptive to the pipeline.

8. Experimental results

We conducted experiments comparing the performance of compiler-parallelized versions of the NAS

benchmarks SP and BT against the hand-written MPI versions of the benchmarks from the NPB2.3b2 release. The compiler-generated parallel codes for these benchmarks were produced by dHPF (our research prototype) and pghpf - a commercial product from the Portland Group. We used different HPF versions of the codes for the pghpf and the dHPF compilers. In particular, each compiler was applied to HPF codes created by the compiler's developers to best exploit its compiler's capabilities. For the dHPF compiler, we created HPF versions of the SP and BT benchmarks by making small modifications to the serial Fortran 77 versions of the benchmarks in the NPB2.3-serial release. We modified the code primarily to add HPF directives and to interchange a few loops to increase the granularity of the parallelization. (More detailed descriptions of these modifications appear below.) With the pghpf compiler, we used HPF versions of the SP and BT benchmarks that were developed by PGI. The PGI HPF implementations are derived from an earlier version of the NAS parallel benchmarks written in Fortran 77 using explicit message-passing communication. PGI rewrote these into a Fortran 90 style HPF implementation in which communication is implicit. (The PGI HPF versions were developed before the NPB2.3-serial release was available.)

The experimental platform for our measurements was an IBM SP2 running AIX 4.1.5. All experiments were submitted using IBM's poe under control of LoadLeveler 3.0 on a block of 32 120Mz P2SC "thin" nodes that was managed as a single-user queue. Loadleveler was configured to only run one process per processor at any time. PGI-written HPF codes were compiled using pghpf version 2.2. All codes - the hand-written NPB MPI codes and the codes generated by pghpf and dHPF - were compiled with IBM's xlf Fortran compiler with command line options -O3 -qarch=pwr2 -qtune=pwr2 -bmaxdata:0x60000000. All codes were linked against the IBM MPI user space communication library with communication subsystem interrupts enabled during execution.

Our principal performance comparisons are on 4, 9, 16, and 25 processor configurations because the hand-written codes in the NAS 2.3b2 release require a square number of processors. We also present the performance for PGI and dHPF generated code on 2, 8 and 32 processors for reference. Single-processor measurements (and in some cases 2- and 4-processor measurements) were not possible because the per-node memory requirements exceeded the available node memory on our system.

In the following subsections, we compare the performance of the hand-written MPI against the compiler-generated code for SP and BT. For each of the benchmarks, we present data for Class A and Class B problem sizes (as defined by the NAS 2.0 benchmarking standards). We present the raw execution times for each version and data size of the benchmarks along with two derived metrics: relative speedup, and relative efficiency. Since we were unable to measure single processor execution times, we lack the basis for true speedup measurements. Instead, based on our experience with the sequential and hand-coded MPI versions on SP2 "wide" nodes that contain more memory, we assume that the speedup of the 4-processor version of the hand-coded MPI programs is perfect (which is approximately true). All speedup numbers we present are computed relative to the time of the 4-processor hand-coded versions. Our relative efficiency metric compares the relative speedup of the dHPF and pghpf generated codes against the speedups of the corresponding hand-written versions. This metric directly measures how much of the performance of the hand-coded benchmarks is achieved by the compiler-parallelized HPF codes. As described in Section 3, the hand-written MPI parallelizations of both the SP and BT benchmarks use a skewed block data distribution known as multi-partitioning. This partitioning ensures that each processor has a subblock to compute at each step of a bi-directional line-sweep along any spatial dimensions. This leads to high efficiency because each processor begins working on a subblock immediately as a sweep begins without needing to wait for partial results from another processor. The multi-partitioning distribution is not expressible in HPF, and therefore the HPF

implementations incur added costs in communication, loss of parallelism, or both.

Instead of multi-partitioning, the PGI and Rice HPF implementations of SP and BT use block distributions. We describe the key features of the two HPF implementations here to provide a basis for understanding the compiler-based parallelization approach used by the dHPF and pghpf compiler versions and its overall performance.

8.1 SP

In developing the Rice HPF implementation of SP from the NPB2.3-serial release, our changes to the serial code amounted to 147 of 3152 lines or 4.7%.[1] The breakdown of our key changes is as follows:

- Removed array dimension (cache) padding for arrays **`u, us, vs, ws, forcing, qs, rho_i, rhs, lhs, square, ainv, and speed`**. The cache padding interfered with even distribution of work in the HPF program. Instead, the dHPF compiler automatically pads arrays in the generated code to make all array dimension sizes odd numbers.
- Eliminated the work_1d common block. Variables **`cv, rhon, rhos, rhoq, cuf, q, ue, buf`** were instead declared as local variables where needed in the lhsx, lhsy, lhsz and exact_rhs subroutines.
- Added HPF data layout directives to specify a BLOCK,BLOCK distribution of the common arrays (**`u, us, vs, ws, forcing, qs, rho_i, rhs, lhs, square, ainv, speed`**) in the **`y`** and **`z`** spatial dimensions.
- Added 6 HPF INDEPENDENT NEW directives. In the lhsx, lhsy, and lhsz subroutines, a NEW directive was used to identify two array as privatizable in the first loop nest. In the exact_rhs subroutine, three NEW directives were used to specify **`cuf, buf, ue, q`**, and **`dtemp`** as privatizable in each of three loop nests.
- In the compute_rhs subroutine, we added an outer one-trip loop along with an INDEPENDENT LOCALIZE directive (a dHPF extension to HPF described in section 4.2) for the **`rho_i, square, qs, us, ws`**, and **`vs`** arrays. This directive has the effect of eliminating communication inside a loop for the specified variables by partially replicating computation of these variables inside the loop so that each element of these arrays will be computed on each processor on which it is used.
- Inlined 2 calls to exact_solution in subroutine exact_rhs where our interprocedural computation partitioning analysis was (currently) incapable of identifying that a computation producing a result in a privatizable array should be treated completely parallel.
- Interchanged loops to increase the granularity of computation inside loops with carried data dependences to increase the granularity of computation and communication for two loop nests in subroutine y_solve and 4 loop nests in subroutine z_solve.

The HPF INDEPENDENT directives are not used by the dHPF compiler to identify parallel loops because the compiler automatically detects parallelism in the original sequential Fortran 77 loops. The only reason HPF INDEPENDENT directives were added to any loops in the code was to specify privatizable variables with the NEW directive and variables suitable for partial replication of computation with the LOCALIZE directive. With the data partitionings in the **`y`** and **`z`** dimensions, only partial parallelism is possible in the **`y`** and **`z`** line solves because of the processor-crossing data dependences. The dHPF compiler exploits wavefront parallelism in these solves using non-owner computes computation partitions for some statements; this leads to wavefront computation that includes pipelined writebacks of non-owned data. dHPF applies coarse-grain pipelining within the wavefront to reduce the ratio of communication to computation. As with the hand-written versions, the problem size

and processor grid organization was compiled into the program separately for each instance with dHPF.

The PGI HPF implementation of SP for pghpf is 4508 lines, 43% larger than the NPB2.3-serial version. The PGI implementation uses a 1D block distribution of the principal 3D arrays along the **z** spatial dimension for all but the line solve in the **z** direction. Before the line solve along the **z** axis, the data for the **rsd** and **u** arrays is copied into new variables that are partitioned along the **y** spatial dimension instead. This copy operation involves transposing the data. Next, the **z** sweep is performed locally. Finally, the data is transposed back. The PGI implementation avoids the use of privatizable arrays found in the NAS SP implementations in subroutines lhsy, lhsz, and lhsx. Instead, in the PGI code several loops were carefully rewritten using statement alignment transformations so that intermediate results could be maintained in privatizable scalars instead of privatizable arrays. Applying these transformations to these loops required peeling two leading iterations, two trailing iterations, and writing custom code for each of the peeled iterations. Such transformations are tedious for users to perform. Even more important, in general it is not possible to always eliminate privatizable arrays with alignment transformations. Handling privatizable arrays efficiently in such cases was a key motivation for the computation partitioning for array privatizables described in Section 4. For the PGI code, only a single version of the executable was used for all experiments as per the methodology used for reporting PGI's official NAS parallel benchmark measurements.

Table 8.1. Comparison between hand-written MPI, dHPF and pghpf for SP

# procs	Execution Time (seconds)						Relative Speedup*						Relative Efficiency**			
	hand-written		dHPF		PGI		hand-written		dHPF		PGI		dHPF		PGI	
	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B
2	-	-	820	--	1213	--	-	-	2.12	-	1.43	-	-	-	-	-
4	436	2094	454	1935	695	2312	4	4	3.84	4.32	2.50	3.62	.96	1.1	.63	.91
8	-	-	259	1086	382	1252	-	-	6.73	7.71	4.56	6.69	-	-	-	-
9	209	783	273	918	381	1296	8.34	10.69	6.38	9.12	4.57	6.46	.76	.85	.55	.77
16	132	466	198	572	222	754	13.21	17.97	8.81	14.64	7.85	11.10	.67	.81	.59	.62
25	88	308	149	459	198	638	19.82	27.19	11.70	18.24	8.81	13.12	.59	.67	.44	.48
32	-	-	127	381	136	508	-	-	13.73	21.98	12.82	16.48	-	-	-	-

*Speedups are relative to the 4-processor hand-written code, which is assumed here to have perfect speedup.
**Relative efficiency is computed by comparing relative speedup of dHPF and PGI generated codes against that of the hand-written version.
-Numbers are not available because the hand-written code requires a square number of processors.
--Numbers are not available because of insufficient per-node memory.

Table 8.1 compares the performance of the hand-written MPI code with the dHPF-generated code and the PGI generated code for both class A and class B problem sizes. For SP, the class A problem is 64^3, and the class B problem size is 102^3. In all cases, the speedups and efficiencies are better for all three implementations on the class B problem sizes. For the larger problem size, the communication overhead becomes less significant in comparison to the computational cost which leads to better scalability. We

look to the relative efficiency measures to evaluate how closely the PGI and dHPF-generated codes approximate hand-coded performance. These measures show that the dHPF compiler is generating computational code that is quite efficient since, on 4 processors, it achieves within 4% of the efficiency of the hand-written MPI for the class A problem size, and is 10% better for the class B problem size. On the class A problem size with the PGI compiler, there is a substantial gap between its efficiency and hand-coded performance. However, this gap narrows substantially for the class B problem size. In comparing the performance with dHPF and the PGI compilers, the efficiency of the dHPF-generated code was uniformly better for both class A and class B problem sizes. As the number of processors is scaled for a fixed problem size, the advantages of the multipartitioning in the hand-coded version become more pronounced. For 25 processors, the efficiencies of both of the HPF implementations drops to nearly half, although the dHPF code is more efficient by a margin of 15% for the class A problem size and this gap increases to 19% for the class B problem size.

To illustrate the performance benefits provided by multipartitioning in the hand-written MPI code, Figures 8.1 and 8.2 show space-time diagrams of 16-processor execution traces for a single timestep of for the hand-coded and dHPF-generated codes respectively. Each row represents a processor's activity. Solid green bars indicate computation. Blue lines show individual messages between processors. White space in a processor's row indicates idle time.

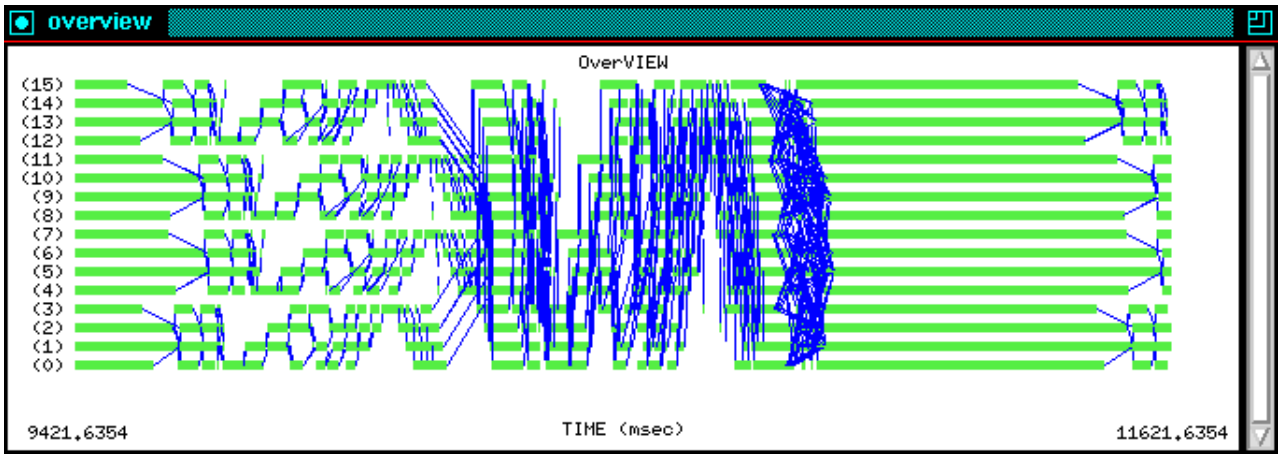
At the left of Figure 8.1, the first 4 bands of blue correspond to the communication in the z_solve phase. The next blue band shows the communication in copy_faces which obtains all data needed for compute_rhs. The next 4 bands of communication correspond to x_solve, and the 4 after that to y_solve. The right of the figure shows the z_solve phase for the next timestep. Figure 8.1 shows that the hand-written code has nearly perfect load balance and very low communication overhead using the multi-partitioning.

Figure 8.1. Space-time diagram of hand-coded MPI for SP (16 processors).

The leftmost 40% of Figure 8.2 shows the communication for the y_solve phase. The central portion shows the z_solve communication. The wide band about 70% across the figure is the communication for compute_rhs, which is largely resembles the communication in copy_faces in the hand-coded version. The long bands of computation near the right side of the figure belong to compute_rhs and x_solve, which is a totally local computation for the 2D data distribution along the **y** and **z** dimensions. Clearly, the largest loss of efficiency is in the wavefront computations of the y_solve and z_solve phases. In

each of the phases, there are two forward pipelined computations, and two reverse pipelines. There are several notable inefficiencies evident in this version of the dHPF-generated code. First, it is clear that the pipelines are at different granularities. The leftmost pipeline is quite skewed: processor 0 finishes its work before processor 2 begins, and similarly for each of the groupings of 4 processors. For this pipeline, the granularity is clearly too large, leading to a loss of parallelism. Unfortunately, dHPF currently applies a uniform coarse-grain pipelining granularity to all the loop nests in a program. An independent granularity selection for each loop nest would lead to superior results. Second, between the two forward pipelines in y_solve, communication occurs in the direction opposite the flow of the pipeline which causes a considerable delay before the start-up phase of the second pipeline. This communication is unnecessary and will in the future be eliminated by a less conservative version of our data availability analysis. Overall, however, while the pipeline granularity and performance can be adjusted, it is clear that multipartitioning provides a far better alternative.

Figure 8.2. Space-time diagram of dHPF-generated MPI for SP (16 processors).



8.2 BT

The Rice HPF version of BT is derived from the NPB2.3-serial release. Our total changes to the serial code amounted to 226 of 3813 lines which is about 5.9%.[1] The changes we made include:

- Removed array dimension(cache) padding for array **`u, us, vs, ws, forcing, qs, rho_i, rhs, lhs, square, ainv`**, and **`speed`**. The padding interfered with even distribution of work in the HPF program. Automatic padding of the generated code is performed by the dHPF compiler.
- Eliminated the work_1d common block. Variables **`cv, cuf, q, ue, buf`** were instead declared as local variables where needed in the exact_rhs subroutine.
- Added HPF data layout directives to specify a 2D or 3D BLOCK distribution of the common arrays (**`u, us, vs, ws, fjac, njac, forcing, qs, rho_i, rhs, lhs, square, ainv, speed`**) in the **`x`**, **`y`** and **`z`** spatial dimensions.
- Added 9 HPF INDEPENDENT NEW directives. In the x_solve_cell, y_solve_cell, and z_solve_cell subroutines, NEW directives were used to introduce two privatizable arrays as temporary variables in two loop nests. In the exact_rhs subroutine, three NEW directives were used to specify **`cuf,buf,ue,q`**, and **`dtemp`** as privatizable in each of three loop nests.
- In the compute_rhs subroutine, we added an outer one-trip loop along with an INDEPENDENT LOCALIZE

directive (a dHPF extension to HPF described in section 4.2) for the **`rho_i, square, qs, us, ws`**, and **`vs`** arrays. This directive has the effect of eliminating communication inside a loop for the specified variables by partially replicating computation of these variables inside the loop so that each element of these arrays will be computed on each processor on which it is used.

- Inlined 3 calls to exact_solution in exact_rhs where our interprocedural computation partitioning analysis was (currently) incapable of identifying that a computation producing a result in a privatizable array should be treated completely parallel.

Table 8.2. Comparison between hand-written MPI vs. dHPF and PGI generated code for BT

# procs	Execution Time (seconds)						Relative Speedup*						Relative Efficiency**			
	hand-written		dHPF		PGI		hand-written		dHPF		PGI		dHPF		PGI	
	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B	Class A	Class B
4	650	--	609	--	590	--	4	--	4.27	--	4.41	--	1.07	--	1.10	--
8	-	--	322	--	318	--	-	--	8.07	--	8.18	--	-	--	-	--
9	304	--	334	--	315	--	8.56	--	7.79	--	8.26	--	.91	--	.96	--
16	181	715	182	727	171	814	14.33	16	14.28	15.75	15.21	14.06	1.00	.98	1.06	.88
25	117	461	143	534	151	632	22.17	24.85	18.21	21.44	17.25	18.11	.82	.86	.78	.73
27	-	-	137	451	151	503	-	-	18.99	25.40	17.26	22.74	-	-	-	-
32	-	-	108	401	102	508	-	-	24.01	28.54	25.49	22.52	-	-	-	-

*Speedups are relative to the 4-processor hand-written code for Class A and 16-processor hand-written code for Class B, which are assumed here to have perfect speedup.
**Relative efficiency is computed by comparing speedup of dHPF generated code with its hand-written counterpart.
-Numbers are not available because the hand-written code requires a square number of processors.
--Numbers are not available because of insufficient per-node memory.

Performance data for both generated codes by dHPF and PGI, and the hand-written code are shown in Table 8.2. The format of the table is similar to that of Table 8.1 for SP. For all three versions, we include data for both Class A and B problem sizes, which are 64x64x64 and 102x102x102 respectively. All speedups for class A are relative to to the 4-processor hand-tuned version, while those for class B are relative to the 16-processor hand-tuned version. As the data shows, the hand-tuned code demonstrated almost linear speedup up until 25 processors, Our code performs extremely well up until 16 processors for both Class A and B. PGI code has excellent performance for Class A up until 16 processors. Our code outperforms the PGI code for Class B executions and shows better scalability on large number of processors. Efficiency and speedup start to decline from 25 processors for both our code and PGI code because the the wavefront parallelism that we realize for the code using a HPF 2D or 3D block data distribution and the 3D transpose that PGI uses have significant overheads, particularly at higher numbers of processors. As in SP, the sophisticated multipartitioning data distribution strategy used in the hand-tuned code [9] achieves much better scalability, but unfortunately is not expressible in HPF. This can be seen from Figures 8.3 and 8.4, which show space-time diagrams of the execution of the

hand-written MPI code and the dHPF-generated code for BT respectively. As with SP, the hand-written code shows excellent load-balance and very low communication overhead. The dHPF-generated code is also much more efficient for BT than for SP, but still has significantly higher overheads due to pipelining than the hand-written MPI.

Figure 8.3. Space-time diagram of hand-coded MPI for NAS BT application benchmark (16 processors).

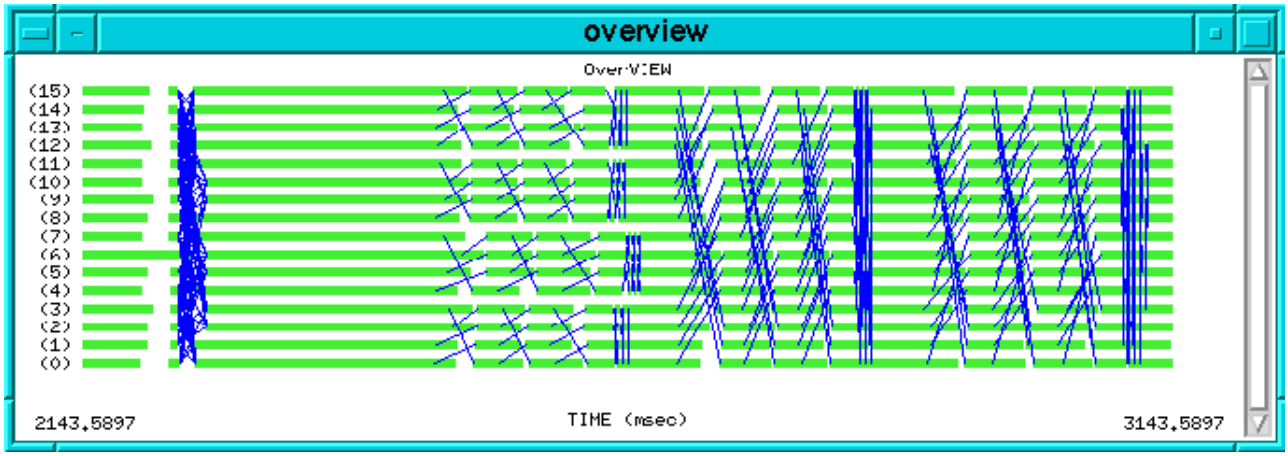
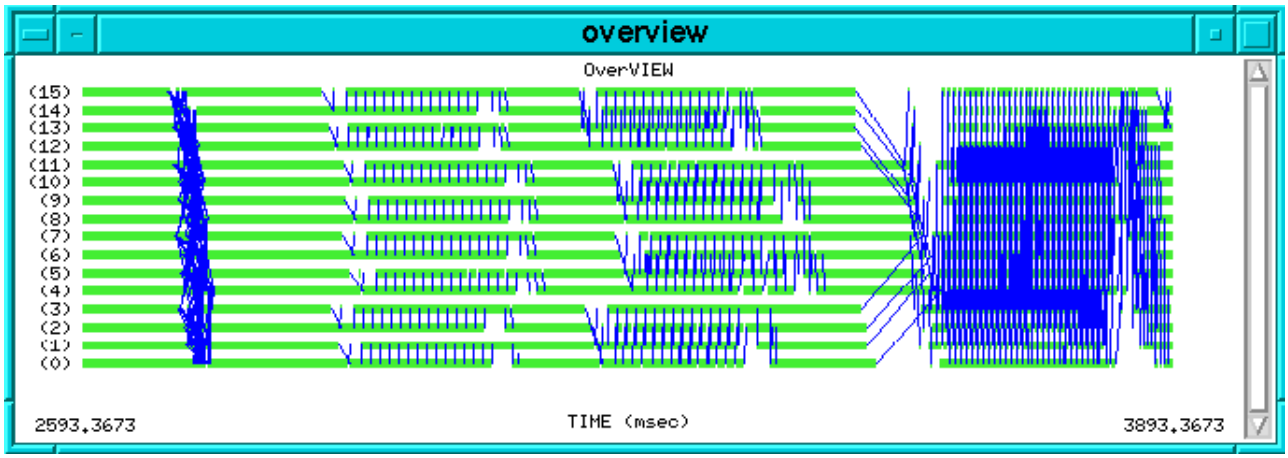


Figure 8.4. Space-time diagram of dHPF-generated MPI for NAS BT application benchmark (16 processors).



9. Conclusions

In this paper, we have presented several compiler optimizations we believe will be important for automatic parallelization of realistic scientific codes in HPF. We have also presented a preliminary performance evaluation of the code generated by the dHPF compiler, and compared this with the performance of hand-written MPI and with code generated by the PGI pghpf compiler. Starting with minimally modified versions of the NAS2.3 serial codes plus HPF directives, our techniques are able to achieve performance within 15% of the hand-written MPI code for BT and within 33% for SP on 25 processors. The code generated by the dHPF compiler also outperforms the code generated by the PGHPF compiler for most cases, even though the HPF code used with dHPF was largely identical to the

original serial version of the benchmarks.

We believe the performance of the dHPF-generated code can be further improved, particularly for SP. There is significant room to improve pipeline performance by automatically choosing the optimal granularity for each pipeline. It is also possible to eliminate a spurious message introducing significant delay between two successive pipelines in each phase. A more complete performance evaluation of the optimizations described in this paper is a subject of our current work. Finally, it would be very interesting to examine whether multipartitioning could be automatically exploited by an HPF compiler (without requiring the programmer to express it at the source code level), in order to enable much higher performance for the class of codes that make line-sweeps in multiple physical dimensions.

Acknowledgements

This work has been supported in part by DARPA Contract DABT63-92-C-0038, the Texas Advanced Technology Program Grant TATP 003604-017, an NSF Research Instrumentation Award CDA-9617383, NASA-Ames Grant No. NAG 2-1181, and sponsored by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA and Rome Laboratory or the U.S. Government.

Footnotes

[1]The measurements of lines modified in the serial codes to create our HPF versions do not include code for procedure interface blocks. Interface blocks are technically necessary for the HPF codes, but we did not include them because dHPF does not yet support them.

References

1 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* (June 1998).

2 V. Adve and J. Mellor-Crummey, Advanced Code Generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, D. Agarwal and S. Pande, Eds. Springer-Verlag Lecture Notes in Computer Science (to appear).

3 D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow, The NAS Parallel Benchmarks 2.0, AMES Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

4 file://ftp.pgroup.com/pub/HPF/examples

5 P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy and

E. Su, The Paradigm Compiler for Distributed-Memory Multicomputers. *Computer 8*, 10 (Oct. 1995) pp. 37-47.

6 S. Chakrabarti, M. Gupta and J-D. Choi, Global Communication Analysis and Optimization. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation* (May 1996).

7 S. Hiranandani, K. Kennedy, and C-W Tseng, Preliminary Experiences with the Fortran D Compiler. In *Proceedings of Supercomputing '93* (Nov. 1993), Association for Computing Machinery.

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

9 V.K. Naik, A Scalable implementation of the NAS Parallel Benchmark BT on Distributed Memory Systems, *The IBM Systems Journal* 34(2), 1995.

10 C.S. Ierotheou, S.P. Johnson, M. Cross and P.F. Leggett, Computer aided parallisation tools (CAPTools) - conceptual overview and performance on the parallelization of structured mesh codes, *Parallel Computing* 22 (1996) 163-195.

11 A. Sawdey and M. O'Keefe, Program Analysis of Overlap Area Usage in Self-Similar Parallel Programs, In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing* (August 1997).