Design and Evaluation of a Computation Partitioning Framework for Data-Parallel Compilers *

Vikram Adve Guohua Jin John Mellor-Crummey Qing Yi

1 Introduction

In a parallel program, a computation partitioning (CP) for a statement specifies which processor(s) must execute each dynamic instance of the statement. The selection of good computation partitionings for the statements in a program can have a dramatic impact on the program's parallel performance. Although researchers have previously developed sophisticated algorithms for CP selection in parallelizing compilers [5, 3, 7, 1], they have paid much less attention to the capabilities required in the rest of the compiler in order to support complex partitionings. In particular, within a parallelizing compiler, the *class of possible computation partitionings* that can be represented has a fundamental impact on the parallel optimizations the compiler can support, and hence on the potential performance of generated code.

In this paper, we present the design and evaluation of a flexible computation partitioning framework used in the Rice dHPF compiler for High Performance Fortran. Our CP framework supports a more general class of static computation partitionings than previous data-parallel compilers, enables sophisticated partitionings that maximize parallelism in the presence of arbitrary control flow, and supports several novel optimizations that have proven essential for obtaining high overall performance when parallelizing scientific programs.

In earlier work, we have shown that the dHPF compiler is able to effectively parallelize HPF versions of existing Fortran codes and achieve speedups that are comparable with hand-coded parallel performance [2, 1]. For example, code generated by dHPF for the NAS application benchmarks SP and BT is within 0–21% of the performance of sophisticated hand-coded message-passing versions of the codes, and these results are achieved with HPF versions that require changes to fewer than 6% of the lines of the original serial codes. Three new CP-based optimizations in the dHPF compiler were key to achieving this level of performance. Two of these three optimizations, along with another new algorithm presented in this paper, require the full generality of our CP framework and could not be implemented in any other existing compiler that we are aware of.

In data distribution-based languages such as Fortran D [13], Vienna Fortran [11], and High Performance Fortran (HPF) [20], it is natural to represent a CP for a statement instance as the set of processor(s) that "own" a particular array element or scalar variable (by virtue of a data distribution). For example, the widely-used *owner-computes rule* [25] (a simple heuristic for computation partitioning selection) can be expressed as the owner of the LHS variable in an assignment. As an approved extension, HPF provides an explicit `ON` directive which a programmer can use to specify the CP for a statement as the owner of a variable, a template element, or a specific processor (explained in more detail in Section 2).

The dHPF compiler generalizes these models to allow static computation partitionings to be expressed as the *union of multiple* `ON HOME` terms. Further, each subscript expression in an `ON HOME` term can be

*Contact Information: {adve,jin,johnmc,qingyi}@cs.rice.edu; Department of Computer Science – MS 132, Rice University, 6100 Main, Houston, TX 77005-1892

a symbolic range with affine bound expressions. Finally, dHPF directly supports loops that contain statements with different computation partitionings and uses a sophisticated code generation strategy to generate efficient code. This overall capability is much more general than previous compilers we know of. To our knowledge, SUIF supported the broadest class of partitionings among previous compilers: it allowed the same partitionings as dHPF except that it did not allow CPs equivalent to general multiple `ON HOME` terms, and it required all statements in a loop to have the same partitioning [4]. Most other previous compilers generally use the owner-computes rule except for certain special cases like reductions and assignments to privatizable variables. They do not allow multiple `ON HOME` terms, and use loop distribution when they encounter statements with different partitionings in the same loop.

Overall, this paper makes the following contributions:

- We describe the design of our general CP model, and in particular the rationale behind the features of the model summarized above.
- We present an algorithm for computing computation partitionings that maximize parallelism in the presence of arbitrary control flow. In contrast, only one previous paper we know of describes in any detail how to partition code in the presence of control-flow [17]. That work uses an ad hoc technique that is more restricted than ours and yields lower parallelism in cases where it does apply (these differences are discussed in detail in Section 5).
- We identify a coupling between insertion of communication, partitioning of communication, and partitioning of control-flow. We describe an iterative algorithm that ensures the correctness all three operations.
- Finally, we evaluate the impact of the CP framework on the performance of key benchmarks, *by examining the performance impact of three optimizations in dHPF that leverage our CP framework*. We give a brief summary of each algorithm (since these have been presented before) and describe how each optimizations exploits the capabilities of the CP framework. We then examine the impact of the individual optimizations on the three NAS application benchmarks, SP, BT, and LU. Our results show that these optimizations are essential to obtain high performance. In fact, we show that the lack of any one of them can hurt performance *by an order of magnitude or more* on message-passing systems (or can require extensive and undesirable restructuring of the source code to achieve acceptable performance).

We begin by providing some background on computation partitionings in data-distribution based languages like HPF. Section 3 then describes the design and key features of the CP framework, including the algorithms for handling control-flow and the interaction between communication placement and CP selection. Section 4 describes the evaluation of the performance impact of the CP framework. Finally, Section 5 discusses related work and Section 6 presents a our conclusions from this work.

2 Background and Motivation

In data distribution-based languages such as HPF, the key arrays of a program are partitioned across the processors of a parallel system in a manner so that parallelism can be naturally extracted by partitioning the computation according to data. For such languages, the *owner-computes rule* [25] defines a simple, natural heuristic for partitioning the computation. The rule specifies that (the value of) a data element is computed by the processor owning that element. This rule provides fairly robust performance for relatively uniform loop nests in data-parallel codes, as long as special patterns like reductions and computations involving privatizable variables are handled differently [21, 19, 16, 10]. Nevertheless, the rule is not optimal

in general [14]. In particular, for loop nests with complex patterns of data reuse (due to data dependences), the owner-computes rule does not perform well and more sophisticated CP selection is required [1].

HPF provides a somewhat more general `ON` directive that a programmer can use to suggest a computation partitioning for a statement or block of statements. This directive can take one of the forms `ON HOME (home-ref)`, `ON PROCESSOR (proc-expr)`, and `ON TEMPLATE (templ-expr)`. These respectively specify a set of processors as the owner(s) of the reference `home-ref`, the processor `proc-expr`, or the owner(s) of the template element `templ-expr`. (A TEMPLATE in HPF is a cartesian arrangement of virtual processors, which is then mapped to a physical processor array using the `DISTRIBUTE` directive.) The `ON HOME` form is the most general because it can be used to represent either of the other two. Its `home-ref` is restricted to a single scalar or array reference.

The goal of our work is to provide a more general framework for computation partitioning that preserves the data-driven approach in such languages, but which permits much more sophisticated choices of computation partitionings. A related goal is to support high-performance partitioning strategies for common constructs like control-flow, privatizable arrays (as well as scalars), data reuse patterns within complex loop nests, and parallel procedure calls within partitionable loops. Early in the design of dHPF, we chose a computation partitioning framework to meet the first goal. This framework has proven capable of supporting a number of aggressive partitioning algorithms that we subsequently developed to provide high performance for these common constructs. The design and evaluation of this framework are the focus of the rest of this paper.

3 The Computation Partitioning Framework in dHPF

The computation partitioning framework in dHPF consists of the following components:

- A general class of partitionings, designed to support aggressive CP selection techniques for data-distribution-based parallel codes.
- A simple but aggressive CP selection algorithm for individual procedures that attempts to minimize the frequency of communication or synchronization.
- An algorithm to preserve correctness and maximize parallelism in the presence of arbitrary control flow.
- An iterative algorithm to assign CPs to communication code and analyze what additional communications is introduced because of these CPs.

These components are described in more detail in the following four subsections.

3.1 Partitionings Supported by the Framework

To motivate the class of partitionings supportable in dHPF, we make the following observations. First, as explained earlier, in a data- distribution-based language, a natural computation partitioning for many ordinary statements is simply the *owner (or home) of some data reference* in that statement (defined formally below). For a single array statement in isolation, one of these choices is in fact optimal [14]. Second, for compound statements such as DO or IF, all processors assigned to any enclosed statement must also execute the compound statement. This means the compound statement must get a CP that is (at least) the union of the CPs of all enclosed statements. This potentially requires a CP to be defined in terms of *multiple home references*. Such CPs also play an important role in optimizations described in Section 4 that replicate computation to reduce communication overhead. Finally, a DO loop, a communication statement, or a

```
!  SEND for references RZ(i-1,j)    !  ON HOME RZ(0:N-1,1:M)
!  RECV for references RZ(i-1,j)    !  ON HOME RX(1:N,1:M) ⋃ RY(1:N,1:M)
do j = 1, M                         !  ON HOME RX(1:N,1:M) ⋃ RY(1:N,1:M)
  do i = 1, N                       !  ON HOME RX(1:N,j) ⋃ RY(1:N,j)
    if (RX(i,j) .lt.  ABS(RXM)) then!  ON HOME RX(i,j)
      goto 100                      !  ON HOME RX(i,j)
    endif
    RXM = RX(i,j) + RZ(i-1,j)       !  ON HOME RX(i,j)
100 if (RY(i,j) .lt.  ABS(RYM)) then!  ON HOME RY(i,j)
      goto 200                      !  ON HOME RY(i,j)
    endif
    RYM = RY(i,j) + RZ(i-1,j)       !  ON HOME RY(i,j)
200 continue
  enddo
enddo
```

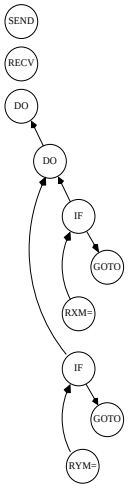


Figure 1: Example illustrating CP propagation and CP assignments for communication. The figure on the right shows the CP dependence graph described in Section 3.3.

function call that computes a range of array elements may each require a home reference with *range-valued subscript* in one or more subscript positions. We allow all these three capabilities in our CP representation.

Assume that every program variable has a unique data layout at each point during an execution of a program (this is true in HPF). For a statement enclosed in a loop nest with index vector $\underline{i}$, and for some variable A, the CP `ON HOME`$A(f(\underline{i}))$, specifies that the dynamic instance of the statement in iteration $\underline{i}$ will be executed by the processor(s) that own the array element(s) $A(f(\underline{i}))$. This set of processors is uniquely specified by subscript vector $f(\underline{i})$ and the layout of array A at that point in the execution of the program.

Any CP in dHPF can be expressed as the union of one or more `ON HOME` terms: $\cup_{j=1}^{j=n}$`ON HOME`$A_j(f_j(\underline{i}))$. Each subscript expression in the subscript list $f_j(\underline{i})$ can be an arbitrary affine expression of loop index variables and other symbolic integer variables. It can also be a symbolic range whose bounds are such affine expressions. The variables A_j may be actual program variables or synthetic (i.e., compiler-generated) variables with synthetic data layouts. For example, we express an `ON TEMPLATE` or `ON PROCESSOR` partitioning using a synthetic variable conforming in shape and distribution with the relevant template or processor array.

To illustrate, consider the HPF source code fragment of Figure 1 which will be used as a running example in this section. This fragment is taken from the SPEC92 benchmark TOMCATV, but the references to array *RZ* are added to illustrate CPs for communication in Section 3.4. All the `ON HOME` CPs as well as the placement of the SEND and RECV operations are chosen by the compiler. (Informally, the communication placement is chosen based on finding the outermost loop level within which the data is not modified before being used, i.e., having no carried dependences for the non-local references or their subscripts.) The statements within the loops are assigned CPs with single `ON HOME` terms (note that not all statements in the loop nest have the same CP). Then, through the CP propagation algorithm of Section 3.3, the enclosing loops are assigned CPs that are union of the CPs of enclosed statements, but vectorized over the ranges of the internal loop index variables. (By unioning two CPs or by vectorizing a CP over a loop range, we *increase* the number of processors that will execute the same statement instance, i.e., we replicate the computation and hence decrease parallelism. Note, however, that a union CP assigned to a loop header does not decrease parallelism for the statements within the loop body.)

One key challenge in supporting such a general class of computation partitionings lies in performing communication analysis and code generation. The dHPF compiler uses an abstract integer set framework

```
SelectCPsForProcedure(Procedure P)
    for (all references R in P)
        ChooseCommLevel(R)   // dependence-based communication placement
    for (each loop L in the program bottom-up)
        PrimaryStmtList(L) = [ s_1 ... s_n :  s_i is a primary stmt in L ]
        for (each S ∈ PrimaryStmtList(L))
            StmtCPChoices(S) = ComputeStmtCPChoices(s)
        LoopCPChoices(L) = { [cp_1 ... cp_n] :  cp_i ∈ StmtCPChoices(s_i), 1 ≤ i ≤ n }
        for (each combination C ∈ LoopCPChoices(L))
            for (each reference r in L)
                if (ReferenceIsNonlocal(r, C))
                    commCost(C) += CommunicationCost(r, C)
        Cmin = C ∈ LoopCPChoices(L) s.t. C has minimum communication cost
        for (each s_i ∈ PrimaryStmtList(L))
            CP(s_i) = Cmin[i]

CPChoiceSet ComputeStmtCPChoices(stmt S)
    if (S is an I/O statement) return  ON PROCESSOR 0
    if (S contains a procedure invocation) return  ON PROCESSOR (0:P-1)
    Refs(S) = { r :  r is a non-privatizable reference in S }
    return {ON HOME r : r ∈ Refs(S)  ∧ HOME(r) ≠ HOME(r')∀r' ∈ Refs(S), r' ≠ r}
```

Figure 2: Intraprocedural CP Selection Algorithm in dHPF

to perform these functions [2]. Any communication analysis or code generation algorithm based on the framework supports any CP in this class.

3.2 Intraprocedural CP Selection Algorithm

The basic intraprocedural CP selection algorithm in dHPF is shown in Figure 2. This algorithm chooses CPs for all *primary statements*, defined as assignments to non-privatizable variables, CALL statements, and I/O statements. In this base algorithm, any statements containing procedure invocations to non-intrinsic procedures are fully replicated (see Section 4.6). I/O statements are assigned `ON PROCESSOR` 0. All other statements are called *auxiliary statements*, and are assigned CPs by the CP propagation algorithm described in Section 3.3.

The CP selection algorithm operates one loop at a time, in bottom-up order on the loops in the program. Within each loop, the algorithm constructs a number of different combinations of CP choices for the statements within the loop, and selects the combination that minimizes the cost of communication induced.

The algorithm begins by using a standard dependence-based communication placement algorithm (not shown in the figure) to choose the outermost loop-level at which communication can occur for *each reference in the program*. This placement level determines the frequency of communication for that reference, *if* the reference is non-local due to an assigned CP. This placement level is independent of the CP choices being considered, and hence needs to be done only once at the start of CP selection. This is essential to make the communication cost evaluation efficient.

For each loop (in bottom-up order), the algorithm operates as follows. It constructs a set of CP choices for each primary statement in the loop, consisting of all references to non-privatizable variables in the statement (eliminating duplicate choices that have identical data layouts). It then considers all combinations of CP choices for the statements in the loop. This is important in order to account for data reuse between different

statements in the same loop nest. It performs a simple communication evaluation (not shown in the figure) to evaluate the cost of each combination based on the frequency of communication, and chooses the minimum cost combination.

Although this algorithm is potentially exponential in the number of statements in a loop, we can use several heuristics to ensure that the running time is reasonable. First, we have found it essential to carefully eliminate duplicate CP choices from the candidate set for each statement. Second, as a heuristic we group statements that have identical sets of choices into a single statement group with a single set of choices, and assign the same CP to all statements in the group. Third, many of the cases where there are intra-loop data reuse (due to loop-independent true data dependences and/or privatizable arrays) are handled more efficiently by the optimization algorithms studied in Section 4. With these heuristics, we have not found this algorithm to be a significant bottleneck to compile-time, even for codes with large loop nests. Furthermore, the last two heuristics handle most of the key forms of data-reuse that occur within a loop. The main remaining form of reuse is due to input dependences (i.e., read references that access the same location), and these could be handled by grouping these statements and assigning them identical CPs, just as in the first heuristic above. Then, given these heuristics, we can entirely avoid considering any combinations of CPs for different statements.

3.3 CP Propagation for Conditional Control Flow

After CPs for primary statements have been selected using the algorithm above, a CP propagation algorithm in dHPF selects CPs for auxiliary statements (defined earlier). This algorithm has two primary goals. The first and most important goal is to maximize parallelism in the presence of arbitrary control-flow while still preserving program correctness. In order to achieve this, we must assign a minimal set of processors (CP) to execute each control-flow statement in a program. The second goal is to guarantee that the values of all scalar privatizable variables are computed on the processors that will use those values, in order to avoid expensive replicated computation or fine-grain communications for those variables.[1]

Both these CP selection steps are performed using a general framework that can be used to propagate CPs from (arbitrary) primary statements to (arbitrary) auxiliary statements. The framework uses a graph we call the CP dependence graph to represent the pairs of statements between which CPs must be propagated. The nodes of the graph are all the statements in the procedure. We add an edge $s_i \rightarrow s_j$ if CP(s_j) must be computed from CP(s_i). The framework then propagates CPs bottom-up on the graph, correctly handling cycles and propagation of CPs between statements in different loops. These two steps (graph construction and CP propagation) are described in turn below.

The function `BuildCPDependenceGraph(CDG, SSA, Priv)` shown in Figure 3 constructs the CP dependence graph for the two propagation goals above. The function uses the well-known Control Dependence Graph (CDG) and Static Single Assignment (SSA) form of programs [12]. The function `BuildCPDependenceGraph` operates as follows.

Intuitively, a structured branch statement (DO or IF) must be executed by all processors that are assigned to execute any enclosed statement. More generally, for any branch statement b, consider any statement s (other than a GOTO statement) that is directly control dependent on b ($b \rightarrow s \in E_{CDG}$). We have to ensure that all processors in $CP(s)$ execute an instance of s *iff* that instance of s would be executed in the original program. Therefore, all processors in $CP(s)$ must also execute b. Therefore, we add the edge $s \rightarrow b$ to CPDG for each edge $b \rightarrow s \in E_{CDG}$. For example, in the loop nest of Figure 1, the assignment to *RXM(i,j)*

[1] This is a simple heuristic that could be improved to select between computing privatizable variables where they will be used or on the processors where the operands are available. E.g., the IBM xlHPF compiler performs this selection, as described in Section 5.

Inputs: $CDG = (N_{CDG}, E_{CDG})$ = Control dependence graph for a procedure
$SSA = (N_{SSA}, E_{SSA})$ = SSA graph for a procedure
Priv(L) = set of variables that are privatizable at loop L, $\forall$ L.
CP(s) = CompPart for stmt s, where s is a primary statement
Output: CP(s) = CompPart for stmt s, for all s in the procedure

PropagateCompParts(CDG, SSA, Priv)
```
CPDG = BuildCPDependenceGraph(CDG, SSA, Priv)
R = Root(CPDG)
PropagateToNode(R, /*currentSCC*/NULL)
```

BuildCPDependenceGraph(CDG, SSA, Priv)
```
N = N_CDG // CPDG = (N, E)
E = Reverse(E_CDG)
for (each GOTO stmt g) // Restore original direction for edges for GOTO stmts
   for (each edge Edge(b,g) ∈ E_CDG)
      Replace Edge(g, b) with Edge(b, g) in E
for (each RETURN or EXIT statement r) // Add edges for RETURN and EXIT stmts
   for (each entry node p) E = E ⋃ Edge(p,r)
for (each loop L) // Add edges for privatizable variables
   for (each variable v ∈ Priv(L))
      for (each def node d of v)
         for (each use u of the value computed in d)
            E = E ⋃ Edge(u,d)
```

PropagateToNode(CompPartDependenceCraph CPDG, Node n, NodeSet* currentSCC)
```
NodeSet* sccN = SCCForNode(n)
if (|sccN| > 1 && sccN != currentSCC) // entering a new cycle of 2 or more nodes
   PropagateToSCC(scc); return
currentSCC = sccN // else |SCC| == 1 or currentSCC = sccN anyway.
VisitState[n] = VISITING
for (all edges c ← n ∈ CPDG)
   if (SCCForNode(c) == currentSCC) continue
   if (VisitState[c] == NOT_VISITED && IsAuxStatement(c))
      PropagateToNode(c, currentSCC)
   cpTerm = PropagateFromStmtToStmt(c, n)
   CP(n) = CP(n) ⋃ cpTerm
   VisitState[n] = VISITED
```

PropagateToSCC(NodeSet* scc)
```
LCA = innermost loop that encloses all nodes in scc
sccCP = EMPTY_CP // the common CP of all the nodes must be a function of outer loop level
for (all nodes n ∈ scc)
   if (IsAuxStatement(n)) PropagateToNode(n, scc)
   cpTerm = VectorizeToLoopLevel(CP(n), n, 1 + LoopLevel(LCA))
   sccCP = sccCP ⋃ cpTerm
if (sccCP != EMPTY_CP)
   for (all nodes n ∈ scc)
      if (IsAuxStatement(n)) CP(n) = sccCP
```

```
CompPart* PropagateFromStmtToStmt(C, K)
   If (C and K are enclosed in the same loop) return CP(C).
   O = outermost loop enclosing C but not enclosing K
   o = LoopLevel(o); // outermost loop has level 1.
   return VectorizeToLoopLevel(CP(C), C, O);

CompPart* VectorizeToLoopLevel(cp, C, o)
   c = LoopLevel(C); // Note:  o <= c.
   for (n = c to o step -1)
      replace loop index variable i_n with its range in cp
```

Figure 3: CP propagation for auxiliary statements in dHPF

is control-dependent on the preceding IF statement, and hence the CP of the assignment is propagated to the IF.

An instance of a GOTO in the parallel execution must be executed by all the processors that reach that statement instance. For instance, in the example, the statement `goto 100` must be executed by all processors that execute the immediately enclosing IF, i.e., the GOTO must get the same CP as the IF. Therefore, for a GOTO statement g, we add the edge $b \rightarrow g$ to CPDG for each edge $b \rightarrow g \in E_{CDG}$. Finally, all processors that enter a procedure instance must exit the procedure instance together. Therefore, for any RETURN or EXIT statement r, we simply add the edge $E \rightarrow r$ for each procedure entry node $E \in N_{CDG}$.

Finally, for each scalar variable privatizable on a loop L, we add the edge $d \rightarrow u$ for each def-use pair $(d, u) \in E_{SSA}$. Privatizable array variables also use such edges, but require careful handling of array subscripts to avoid needless replication of computation, as described in Section 4.4.

The mutually recursive functions `PropagateToNode` and `PropagateToSCC` are used to propagate CPs through the graph. Most of the code in these functions is a straightforward bottom-up traversal of the CPDG, invoking `PropagateFromStmtToStmt` to translate CPs for each edge in the CPDG. Here, we only focus on two key aspects of the propagation, namely (a) the handling of cycles, and (b) the propagation of CPs between statements in different loops.

The function `PropagateToSCC` computes CPs for nodes in a cycle, i.e., a strongly-connected component (SCC) in the graph. All nodes in a cycle must be assigned the same CP, and this common CP must only refer to index variables of the loops that enclose all statements in the SCC. Therefore, the function first propagates CPs from nodes outside the SCC to nodes in the SCC, using the function `PropagateToNode`. For each node, it uses `PropagateToLoopLevel` to expand all loop index variables other than those of the common enclosing loops. It then assigns the union of all these CPs to each node in the SCC.

When propagating a CP from statement C to statement K, let O be the outermost loop enclosing C but not enclosing K. Then, index variables for O and inner loops must be replaced by their ranges when assigning CP(C) to K. The function PropagateToLoopLevel performs this function for each CP term in CP(C).

3.4 CP Assignment for Communication

The dHPF compiler can place communication at arbitrary points within the control-flow of a program depending on the dependence or dataflow properties of the code (e.g., to minimize communication frequency or to hide synchronization or communication latency). To preserve a uniform approach for CP propagation and parallel code generation, it is important to assign CPs to communication code just as to computational

```
AssignCommunicationCPs()
    repeat
       FindAndPlaceCommunication()
       for (each communication placeholder statement C)
          AssignCommunicationCP(C) // see text for details
       Propagate communication CPs only using PropagateToNode();
    until (CPs for all statements are unchanged)
```

Figure 4: Iterative CP assignment for communication and control-flow

statements.

In dHPF, CPs are assigned to communications as follows [27]. Consider a set of non-local read references $r_1 \ldots r_M$ for which communication has been coalesced (currently these references must be to the same array). Let the CPs for these references be $cp_j, 1 \leq j \leq M$, and assume the communication can be vectorized out of enclosing loops $i_1 \ldots i_n$. The data must be sent from the processor(s) that own the data elements to the processor(s) executing the reads. Therefore, the SEND placeholder is assigned the CP $\bigcup_{j=1}^{j=M}$ `ON HOME`r_j, except that each expression r_j must be vectorized over the ranges of all the variables $i_1 \ldots i_n$. The RECV placeholder is simply assigned $\bigcup_{j=1}^{j=M} cp_j$, vectorized over the same ranges. (Thus, in the example of Figure 1, the SEND placeholder gets the CP `ON HOME` *RZ(i-1, j)* vectorized over the ranges of i and j. The RECV placeholder gets the union of the CPs of the two assignments where RZ is used, vectorized over the same ranges.)[2] Thereafter, all communication statements are treated uniformly with other statements (as far as CPs are concerned). This simple and uniform support for communication CPs is possible because the dHPF framework permits multiple `ON HOME` terms as well as subscript ranges in the home references.

The CP assigned to communication (i.e., to the placeholder) can change the CPs for enclosing loops or conditional branches. Unfortunately, this CP may then cause variables referenced in the loop bounds or branch expressions to be non-local if the values are not computed or owned by all processors in the new CP. This implies a circular dependence between the communications required for a program, and the CPs required for the communications. (Note that this circular dependence is not specific to our CP framework, and would occur in any compiler that did not use fairly simplistic handling for communication and control-flow.)

In order to preserve the full generality of the CP framework, we use an iterative algorithm to find and place communications, assign and propagate communication CPs, and repeat these two steps until the CPs for all statements remain unchanged. This algorithm is conceptually simple and is shown in Figure 4. The number of iterations is bounded by the maximum nesting depth of DO and IF statements in the current procedure. In practice we have never needed more than 2 iterations, and that is required when communication is placed inside a loop in the same procedure. Then the first assignment of CPs to communication changes the CPs for the enclosing loop, but usually no new communication is detected and so no further changes are made. Nevertheless, the iterative algorithm is essential for correctness.

[2] The CPs assigned by this technique are conservative because not all values accessed by a reference need to be communicated. Expressing a more precise CP, however, can only be done in terms of the partitioned SPMD code and not in terms of source code variables and distributions. Instead, we rely on the communication code generation to ensure that excess communication does not occur.

4 Evaluation of the Impact of the CP Framework

The goal of this section is to evaluate the impact of the CP framework on the performance of key benchmarks, by examining the impact of the major optimizations in dHPF that rely on the CP framework.

In earlier work, we have shown that the dHPF compiler is able to achieve very good overall speedups for several benchmarks including two of the NAS application benchmarks (SP and BT) [2, 1]. For SP and BT, these speedups are comparable to (within 0–21% of) sophisticated hand-coded message passing versions of the codes, even though the latter used more efficient data distributions (which they called multipartitioning [24]) that cannot be expressed in HPF.

Three new optimizations in the compiler that were key to obtaining the above results directly rely on the capabilities of our CP framework. In this section, we evaluate the impact of these individual optimizations on the performance of three NAS benchmark applications (SP, BT and LU). We believe that these experiments give direct evidence for the performance impact of the general CP framework used in dHPF.

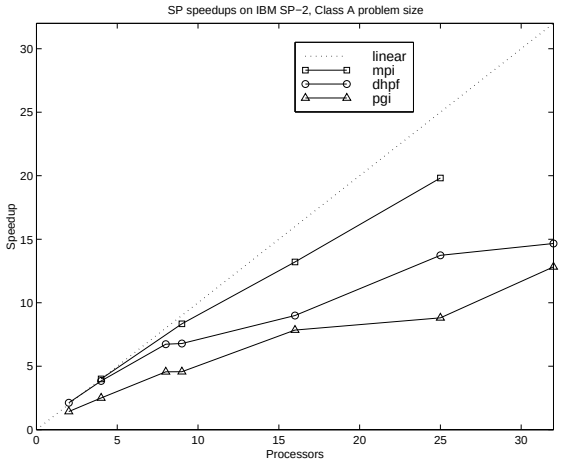
After describing the benchmarks and our experimental methodology, we begin by giving a summary of the overall speedup results for the three NAS application benchmarks. The overall speedups for SP and BT are summaries of those presented in [1], while the results for LU are new. The next three subsections then examine the impact of the individual optimizations. In each case, we include a brief summary of the optimization, describe why the optimization requires the CP framework, and then present the experimental results.

4.1 Benchmarks and Experimental Metholodogy

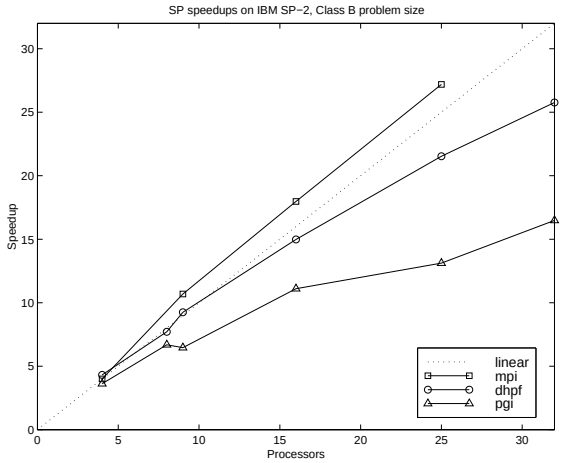
The three NAS application benchmarks (BT, SP and LU) are important benchmarks representative of sophisticated computational fluid dynamics codes [6].. They all solve the same partial differential equation, a 3D discretization of the Navier-Stokes equation, but differ in the factorization of the operator matrix and in the solution techniques used. SP and BT both use different versions of the Alternating Direction Implicit (ADI) integration method, but SP solves scalar pentadiagonal systems while BT uses block-tridiagonal systems of 5×5 blocks. LU uses Symmetric Successive Overrelaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations.

We created HPF versions of these benchmarks by minimal modification of the serial codes in the NPB-2.3 release. (The computations in these serial codes are essentially identical to the computational parts of the MPI codes in the NPB2.3-b2 release, which we use for our comparisons.) The major modifications impacting the measured performance were to add data layout directives, use the NEW directive to mark several arrays as privatizable, interchange a few loops in SP to increase the granularity achieved through coarse-grain pipelining, and inline two procedure calls in SP and three (of many) in BT. The data distributions used in the experiments are `(BLOCK, BLOCK, BLOCK)` for BT, `(*, BLOCK, BLOCK)` for SP, and `(BLOCK, BLOCK, *)` for LU. These modifications affected less than about 6% of each of the codes.

The evaluations were performed on an IBM SP2 running AIX 4.1.5 and using the user-space communication library. The code generated by dHPF was compiled with the local IBM xlf compiler with options -O3 -qarch=pwr2 -qtune=pwr2 -bmaxdata:0x60000000. During the experiments, we ensured that the SP2 nodes used by our job were not shared with any other user jobs.

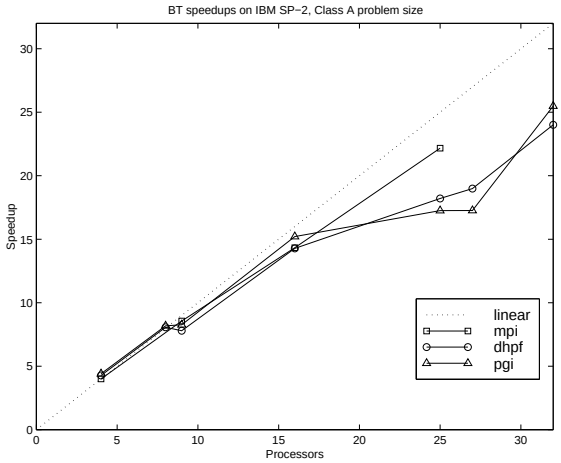


(a) Class A – 64x64x64.

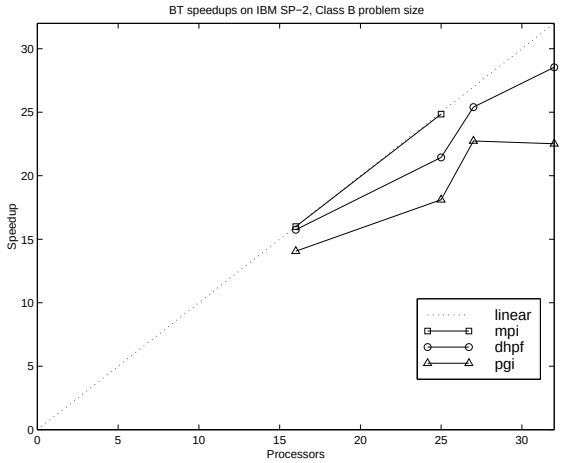


(b) Class B – 128x128x128.

Figure 5: Overall speedups of NAS SP benchmark on IBM SP-2.

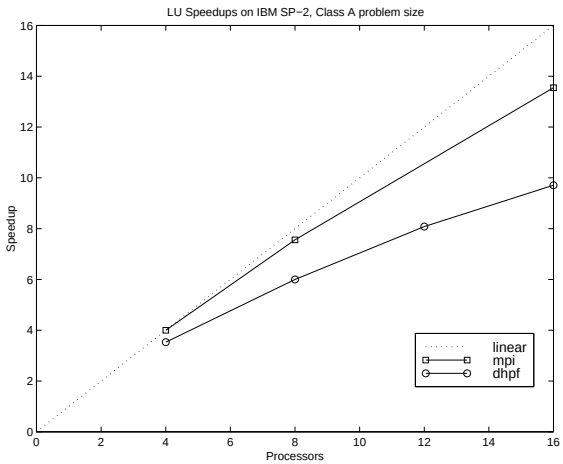


(a) Class A – 64x64x64.

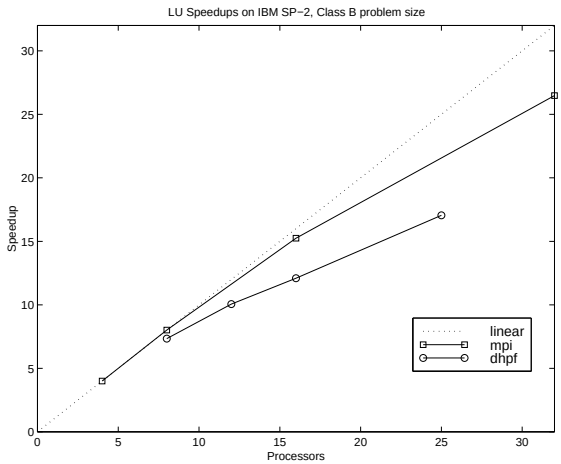


(b) Class B – 128x128x128.

Figure 6: Overall speedups of NAS BT benchmark on IBM SP-2.



(a) Class A – 64x64x64.



(b) Class B – 102x102x102.

Figure 7: Overall speedups of NAS LU benchmark on IBM SP-2.

4.2 Summary of Overall Performance of the Benchmarks

We begin by giving a brief summary of the overall speedups for the SP and BT benchmarks presented previously [1], plus the overall speedups for LU.[3] The dHPF results are for HPF versions we created as described earlier. The pgHPF results are for HPF versions of the benchmarks written by the Portland Group to optimize performance obtained from their compiler. (There is no pgHPF version of LU, probably because their compiler does not support pipelined communication which is essential for this code.) These results are compared against hand-coded message-passing (MPI) versions from the NAS NPB2.3b2 release.

Figures 5, 6 and 7 show the speedups of the SP, BT and LU benchmarks respectively, for the Class A and Class B sizes in each case. While ideally all measurements would be relative to the performance of the sequential code, memory limitations in our system prevented us from measuring performance for these problem sizes on 1, 2 and in some cases even 4 and 8 nodes. Instead, we normalized the speedups for each application and problem size to that of the hand-coded version running on the smallest number of processors feasible and assumed that the hand-coded version achieves perfect speedup for that configuration. It is worth noting that the hand-coded MPI implementations of SP and BT have a significant advantage over dHPF and pgHPF in that they use a multipartitioning strategy [24] for distribution of work and data that eliminates the need for pipelining and results in better load balance than any distributions expressible within HPF. The advantages of the multipartitioning become even more pronounced for higher numbers of processors at a fixed problem size. Multipartitioning is not applicable to LU which uses 3D wavefronts rather than alternating 1D wavefronts. For the SP and BT application benchmarks, the hand-coded MPI programs are only designed for a square number of processors; for LU, the number of processors must be a power of two.

From the figures, we see that performance of the dHPF-generated code falls in between that of the hand-coded MPI and the pgHPF codes in most cases, but is still fairly close to that of the hand-coded MPI, particularly for the larger problem sizes. dHPF does better on BT than on SP, principally because coarser-grain communication makes the computation more efficient. For both these codes, the relative efficiency starts to decline above 25 processors for both dHPF and pgHPF, because the increasing advantage of multipartitioning compared with the wavefront parallelism used by dHPF and the 3D transpose used by pgHPF. In LU, dHPF falls short of the hand-coded MPI principally because the dHPF version has communication inside a subroutine called from within a loop, whereas this communication was hoisted outside the loop in the hand-coded MPI. Overall, with dHPF we were able to achieve performance within 15% of the hand-written MPI code for BT and within 21% for SP (both on 25 processors), and within 21% for LU on 16 processors. Furthermore, dHPF outperforms PGI's pgHPF for most cases even though the HPF code used with dHPF was much more similar to the original serial version of the benchmarks.

4.3 Evaluation of Individual Optimizations

The following three subsections examine the individual impact of three separate optimizations that exploit the CP framework. These are:

- CP selection for privatizable and localizable arrays.
- Integrated loop distribution and CP selection for intra-loop data reuse.
- Interprocedural CP selection for data-parallel codes.

When comparing performance with and without an individual optimization, it is important to ensure that all other applicable optimizations are retained. This is because separate optimizations are rarely independent,

[3] The speedups for SP are actually somewhat improved compared to those in [1] through improved scheduling of asynchronous communication for pipelines and better selection of pipeline granularities.

```
CHPF$ DISTRIBUTE (BLOCK): x, y, z
CHPF$ INDEPENDENT, NEW(p)
do i = i, n
    do j = 1, n
      p(j) = ...                ! ON HOME x(i,j+1), y(i,j-1)
    enddo
    do k = 2, n-1
      x(i,k) = ... p(k-1) ...   ! ON HOME x(i,k)
      y(i,k) = ... p(k+1) ...   ! ON HOME y(i,k)
    enddo
enddo
```

Figure 8: Example for CP propagation for privatizable and localizable arrays. All CPs shown are selected by the compiler.

---

e.g., one optimization may render a second one less useful or conversely may be essential to obtain the benefit of the second one. In these comparisons, therefore, we retain the full set of optimizations performed in the compiler and only turn on or off the optimization being studied. In addition to the three optimizations listed above, the full set includes a number of other optimizations, many of which are particularly important on message-passing systems. All of these optimizations fully support any CP in our CP framework. The abstract integer set framework used in dHPF for data-parallel program analysis and code generation is essential to making this possible [2]. These optimizations include:

- Communication vectorization for arbitrary regular communication patterns.
- Message coalescing for arbitrary affine references to an array.
- Overlap areas for holding non-local data in stencil computations.
- In-place communication that avoids data packing and/or unpacking when the data to be communicated is contiguous in memory.
- A loop-splitting transformation that separates iterations that access non-local data from purely local iterations. This allows the compiler to minimize synchronization delays due to the MPI protocols by overlapping communication latencies with computation *within* a single loop-nest. On the IBM SP-2, we exploit this to post asynchronous receives (MPI_Irecv) before the loop nest begins and asynchronous sends (MPI_Isend) after all purely local iterations of the loop nest have been executed.
- Recognizing reduction operations and implementing them efficiently through careful CP selection and the use of MPI reduction routines [22].
- Coarse-grain pipelining in one or more dimensions. SP and BT require 1-dimensional pipelines because they use ADI solvers, whereas LU requires a 2-dimensional pipeline for its SSOR solver. Currently, the pipeline granularity has to be specified on the command line by the user, and is used for all pipelines in the program.
- A sophisticated control-flow simplification algorithm that works by propagating inequality constraints on integer variables using the control-dependence graph [23].
- Padding distributed array dimensions to ensure that all array dimensions are of odd length.

In the experiments that follow, all of the optimizations in this list were always left on.

Optimization	SP		BT		LU	
	With	Without	With	Without	With	Without
CP Propagation for PrivLoc Arrays	237	3880	192	303	27	900000
Loop Distribution and CP Selection	237	2353	NA	NA	NA	NA
Interprocedural CP Selection	NA	NA	192	391	NA	NA

Table 1: Execution time with and without each optimization (in seconds)
Problem Size: Class A – 64x64x64. Number of time steps: SP – 400, BT – 200, LU – 50.

4.4 CP propagation for privatizable and localizable arrays

Parallel scientific codes frequently use privatizable arrays to hold temporary values within a loop nest.[4] In HPF, such arrays can be identified to the compiler using the `NEW` directive. Such an array, however, may either cause the computation to be fully replicated or may introduce expensive fine-grain communication within the loop (depending on the data layout of the array).

Consider the simple example shown in Fig. 8 (ignoring the `ON HOME` selections for a moment). Suppose that the second dimension of the `x` and `y` arrays is partitioned among several processors. For a processor to execute the iterations of the `k` loop for the elements of `x` and `y` it owns, the processor will require the values of the privatizable array `p` that are referenced within those iterations. There are three principal choices for partitioning the loop that computes the array `p`: (1) Fully replicated: this may require expensive communication for any partitioned right-hand side values used in computing `p`. (2) Partitioned according to some data layout for `p`, i.e., `ON HOME p(j)`: this requires expensive fine-grain communication between the j and k loops. (3) Selectively replicated so that each processor will compute just the values of `p` it will need to execute its partition of the `k` loop. The dHPF compiler implements the third option by translating the CP(s) of the statement(s) that use the privatizable array and assigning these to the statements that compute the array. In Fig. 8, assuming the `ON HOME` CPs for the assignments to `x` and `y` are as shown, the dHPF compiler computes the CP shown for the assignment to `p`. For example, since `p(k-1)` is used to compute `x(i,k)`, `p(j)` must be computed by the owner of `x(i,j+1)`. Repeating this for both uses of array `p` causes `p` to be computed exactly where it is needed.

A similar need for partially replicating computation can arise for definitions of partitioned arrays that may be live upon loop exit. [5] We support an additional directive, LOCALIZE, for this purpose. LOCALIZE has the same semantics as NEW except that values are permitted to be live after the loop. For CP propagation, the dHPF compiler treats LOCALIZE almost identically to NEW, except that the CP for a definition of a localizable variable is always computed by its owner in addition to whatever processors are specified by propagated CPs.

Need for the CP framework: This optimization for privatizable and localizable arrays relies on the full generality of the CP framework. The CPs selected are usually non-owner-computes CPs. The union of multiple `ON HOME` terms is clearly required when there are multiple uses of the privatizable array. Finally, a range subscript may be required when the same array element is used on all iterations of a subsequent loop. To our knowledge, this optimization could not be implemented in any previous parallelizing compiler.

[4] An array is *privatizable* within a loop when (a) each element of the array used in an iteration of the loop is defined earlier within the same iteration, and (b) the values of the array elements defined within the loop are not used outside the loop.

[5] In particular, some arrays satisfy condition (a) of the definition of privatizable arrays, but may not satisfy (b). We call such arrays *localizable* within the loop.

The propagation of CPs from an array use to a definition directly leverages the propagation strategy in section 3.3. In particular, the propagation of the CPs (transformed as explained above) is performed automatically by simply adding edges in the CP dependence graph from uses to definitions of privatizable or localizable arrays (just as edges for privatizable scalars were added previously).

Performance Impact: Table 1 shows the execution time of the three NAS applications when this optimization is turned on or off. As explained earlier, all other optimizations are retained in order to isolate the additional impact of this optimization on the final performance.

We can see that this optimization improved overall performance by a large factor for all three applications. The performance improvements are so large because, without the optimization, the compiler introduces expensive fine-grain communication (option 2 of the three options listed above). The improvements are particularly large in SP and LU because the optimization eliminates inner-loop broadcast communications (in one loop nest each for subroutines `lhsy` and `lhsz` in SP and subroutines `blts` and `buts` in LU). In BT, the optimization avoided inner-loop communications in all the distributed sweeps of BT which is caused by a temporary variable introduced to enable interprocedural CP propagation.

Because the performance without this optimization is so poor, we examined what alternative the pgHPF compiler would use in such loops. We compiled *our* HPF version of the NAS benchmarks (which uses privatizable arrays just as in the original serial code) using the pgHPF compiler. The total execution times with pgHPF were extremely high for these benchmarks, and a few modifications (particularly for loops with loop-independent dependences) were required to obtain more meaningful times. These modifications and the resulting times are described in subsection 4.5.

For loops that use privatizable arrays, we found that pgHPF used option (1) or (2) above depending on whether the privatizable array was replicated or distributed. In the former case, it required full broadcast of one or more arrays used on the right-hand-side of this computation. In the latter case, it required fine-grain communication within the loop nest. Both of these options are clearly extremely inefficient. The only other viable option we know of is to extensively rewrite the loop by hand (using a realignment transformation) to use privatizable scalars instead of privatizable arrays. This transformation was used in the PGI version of the SP benchmark to obtain the speedups reported in Section 4.2 (it increases the length of each loop nest by a factor of 5). Furthermore, in general it is not always possible to eliminate privatizable arrays with this transformation. In comparison with these options, the general CP framework in dHPF permits a more elegant, high performance solution that works for any use of privatizable arrays and does not require any restructuring on the part of the programmer.

4.5 Combined CP Selection and Loop Distribution

Large loop nests in scientific codes often have substantial intra-loop reuse of array values, which appear as loop-independent data dependences at various loop levels. Without special handling, such dependences will result in communication inside loops which can be very costly. We have developed an algorithm that integrates loop distribution with CP selection to avoid communication inside loops where possible. We use a two-prong strategy. First, if possible, statements connected by loop-independent dependences are constrained to get identical CPs. This approach minimizes the changes to code shape, which preserves any temporal reuse in the original code as much as possible. Otherwise, wherever it is legal, the algorithm distributes the loop to avoid communication inside the loop.

The algorithm works in conjunction with the CP choices used in the CP selection algorithm of Section 3.2. The goal of this algorithm is to group together any statements that are connected by a loop-independent dependence and have at least one common CP choice. Each statement initially belongs to a group consisting only of itself. For each loop-independent dependence connecting statements S1 and S2 belonging to different

groups G1 and G2, the algorithm checks the intersection of the CP choice sets for G1 and G2. If the intersection is non-empty (i.e., there is at least one common CP choice), the algorithm combines G1 and G2 into a single group and assigns the intersection as the new CP choice set for the combined group. If the intersection is empty, the algorithm later does a minimal legal distribution of the loop so that all such pairs of groups are put into different loops if possible. If two such groups cannot be distributed, inner-loop communication for that loop-independent dependence cannot be avoided.

Need for the CP framework: The key requirement for this algorithm is the ability to use non-owner-computes CPs since, typically, the statements at the two end-points of a loop-independent dependence will be assigned some common CP.

This algorithm also depends on the availability of explicit CP choices within the overall CP selection algorithm. It interfaces with the overall algorithm only by invoking ComputeStmtCPChoices() to obtain CP choices for each statement, and by returning a (possibly) more restricted set of choices for each statement. This algorithm is otherwise independent of the overall algorithm, and can be used within any overall algorithm that works with explicit CP choices for statements.

Performance Impact: Table 1 shows that this algorithm improved the execution time of SP by almost a factor of 10, while it has no effect on BT and LU. The algorithm effectively avoided inner loop communication in the distributed sweeps of SP by constraining CP choices so that the end-points of loop-independent dependences were executed by the same processor. Loop distribution was therefore not required in this case. The algorithm could also have been important for BT and LU, but the cases where it would be needed are instead handled by the CP propagation for privatizable arrays, which eliminates inner-loop communication for such arrays.[6] In practice, we believe both these algorithms will be needed to avoid poor performance on different loop nests.

Again, we also examined how the relevant loop nests of SP were handled by the pgHPF compiler. In the version of code used for the dHPF results, the compiler introduced extremely expensive fine-grain communication for the loop-independent dependences that cause the code to take several hours for a single time-step. We then added `INDEPENDENT` directives to the two parallel loops in the loop nest. The compiler then completely replicated the computation in the remaining (non-independent) loop, which avoided the inner-loop communication but introduced two expensive broadcasts of two arrays before the loop nest. The code now required about 89 seconds for a single time-step of the A class benchmark, i.e., about 100x slower than the execution time achieved with dHPF. (In the improved version, the inefficiency due to handling of privatizable arrays described in Section 4.4 was also significant.)

4.6 Interprocedural CP propagation

An essential step in parallelizing a loop containing procedure calls is to select appropriate computation partitionings for the call sites and the called procedures. In data-parallel loops, procedure calls are typically used to compute on some subset of the dimensions of the data domain, e.g., pointwise, column-wise, or planar operations within a 3D domain. We take advantage of this characteristic to develop an efficient interprocedural algorithm to realize data-parallelism across procedure boundaries. The algorithm requires

[6] Conversely, when the CP propagation for privatizable arrays was turned off in the experiments in the previous subsection, one might expect the selective loop distribution optimization to help eliminate the high-overhead inner-loop communication. This does not happen, however, because we do not consider dependences due to privatizable arrays in the selective loop distribution algorithm. Performing loop distribution in these cases would require expanding the privatizable arrays to be full 3D instead of 1D arrays. That greatly increases the memory requirements of these codes, and is not worthwhile when the CP propagation algorithm is available.

no interprocedural context information for each call site, and can simply operate bottom up on the call graph. (For the same reason, it is also very rare that recursion is used to express this form of parallelism, and we have not extended our algorithm to handle cycles in the call graph.)

The algorithm can be thought of as extending the CP propagation algorithm of Section 3.3 so that CPs continue to be propagated upward, out of a procedure and into each of its call sites (using the call graph instead of the CP dependence graph). Note that the intraprocedural CP propagation algorithm assigns a unique CP to each procedure entry point, which we call the procedure CP. Because we operate bottom-up on the call graph, the procedure CP will already have been computed for all of the callees of the current procedure. The key new step is that at each call site within the current procedure, we must translate the callee's procedure CP from the name space of the caller to that of the callee. We then constrain the set of possible CP choices for the invoking statement to a singleton set containing only this CP. This ensures that this CP will finally be selected for the statement. (If there are multiple invocations in a statement, each of the propagated CPs must be unioned into the assigned CP.) We then invoke the separate intraprocedural CP selection and CP propagation algorithms (Figures 2 and 3 respectively) to select CPs for all statements within the current procedure.

Need for the CP framework: The interprocedural algorithm requires the full expressive power of the CP framework in dHPF. In general, an arbitrary non-owner-computes CP may be assigned to any call-site, depending on the CPs chosen for the callee. The union of multiple `ON HOME` terms would be required if there are multiple call-sites within a single statement. (This is over and above any union CP that may occur as the procedure CP for a single call site.) Finally, if a procedure call computes a range of values in one or more dimension of an array, a range subscript may be required in the CP for the call site.

In each of the three NAS benchmarks, both the non-owner-computes CPs and the range CP subscripts occur at various call sites. A range, however, only occurs in non-distributed array dimensions (so that it does not represent multiple processors).

Performance Impact: Table 1 shows that this algorithm improved the execution time of BT by about a factor of 2, but has no effect on SP and LU. In the latter two benchmarks there are in fact procedure calls within parallel loops, but all such occurrences are in the initialization phases of the codes. The algorithm was necessary to parallelize those loop nests, but those phases are not included in the timings.

There are two reasons why the performance improvement in BT is so large. First, the procedure calls occur in some of the key loop nests of the program. Second, the interprocedural CP selection optimization combines with the CP propagation for privatizable arrays in the caller to help vectorize communication out of a loop in the caller. Without interprocedural CP selection, the compiler is forced to either replicate the computation of a privatizable array (requiring very expensive broadcast communication), or to insert communication within the callee (causing expensive fine-grain communication). The compiler currently uses the latter.

5 Related Work

With few exceptions, research and commercial HPF compilers almost exclusively use the *owner-computes* rule [25] to assign CPs to statements [8, 21, 19, 15, 16, 9, 10, 28]. In these compilers, the only exceptions to this rule are for reductions and (in the IBM pHPF compiler [16]) for assignments to privatizable variables. Two compilers that use non-owner-computes CPs in more general cases are decHPF and SUIF. The decHPF compiler will in some cases compute the right hand side of an assignment on a processor other than the owner of the LHS term. However, the description in the literature leads us to believe that such CP choices

are made for single statements in isolation rather than using global information as in dHPF [18]. The SUIF compiler [4] provides the broadest class of CPs but it does not support CPs equivalent to general multiple `ON HOME` terms, and it requires that all statements in a loop have the same computation partition.

Several researchers have described algorithms to select CPs or CPs and array data layouts automatically [5, 3, 7]. These algorithms use more sophisticated techniques for general intraprocedural CP selection than our base algorithm in Section 3.2. They are all restricted, however, to selecting a single CP for an entire loop iteration. In contrast, many of the CP-based optimizations in dHPF (e.g., for privatizable arrays, for integrated CP selection and loop distribution, and interprocedural CP selection) often *require* separate CPs for individual statements within a loop, in order to be effective. In fact, the previous papers do not describe how these common patterns are handled by their algorithms. The previous papers also do not suggest how to select CPs interprocedurally in the presence of procedure calls.

Gupta has described careful strategies for partitioning assignments to privatizable variables and for partitioning control-flow statements, and prototyped these in the IBM pHPF compiler [17]. His algorithm for privatizables chooses between three options: (1) aligning the variable with an array reference on the RHS of the assignment; (2) aligning with the consumer reference (typically the LHS of the statement using the value); and (3) to assign no alignment if no communication would be required. Their algorithm, however, is inherently designed to work with the owner-computes rule used in pHPF, so that "alignment" of the privatizable implicitly determines the selected CP of the assignment. Furthermore, this privatization is not performed if there are multiple uses with different alignments. In contrast, we use a more rigorous and general algorithm that works with arbitrary CP selection algorithms, and can handle multiple unaligned uses of the privatizable data. We have found these capabilities essential for several loop nests in all three NAS application benchmarks, and the pHPF compiler would not be able to perform privatization in these cases.

Gupta's algorithm for partitioning control-flow statements is restricted to the cases where it can use the privatization-without-alignment technique described above (i.e. introducing no CP guards for the control-flow statement), and where the control-flow statement does not transfer control out of the enclosing loop. If either condition is violated, they fully replicate the control-flow. Furthermore, by not using alignment, processors that only execute statements outside an IF (but within the enclosing loop) may also have to execute the IF. In contrast, again, we use a more rigorous and general algorithm based on the control-dependence graph which allows arbitrary CPs of the CP framework, (i.e, not restricted as in privatization-without-alignment), does not have the restriction on control-flow, and selects more precise CPs (i.e., with higher parallelism) for cases such as the example above.

For stencil-based access patterns, the TOPAZ tool [26] provides powerful compiler support to reduce communication frequency by replicating boundary computation, equivalent to our localization optimization of Section 4.4. Their tool operates on explicitly parallel SPMD codes. In contrast, our algorithm applies to non-stencil codes as well, and does so within the context of a high-level parallel language where we require full support for computation partitioning and communication analysis.

6 Conclusion

In this paper, we have described and evaluated a flexible framework for static computation partitioning of data-parallel codes. The framework provides a more general class of static partitionings than previous compilers, which we have found to be essential to express several sophisticated CP optimization algorithms. We have presented experimental results showing that three optimizations that rely on the CP framework are essential for obtaining high performance on HPF versions of the NAS parallel application benchmarks. In fact, with these optimizations, the compiler-generated code approaches the performance of sophisticated

hand-coded MPI versions of these benchmarks, even though the latter used a more effective skewed-cyclic data distribution (multipartitioning) that cannot be expressed in HPF. We believe these results clearly demonstrate the importance of the flexible computation partitioning framework used in the dHPF compiler.

References

[1] 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.

[2] 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.

[3] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, September 1995.

[4] 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.

[5] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[6] 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.

[7] R. Barua, A. Agarwal, and D. Kranz. Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Distributed-Memory Multiprocessors. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.

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

[9] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 704–709, San Francisco, CA, February 1995.

[10] Satish Chandra and James R. Larus. Optimizing Communication in HPF Programs on Fine-Grain Distributed Shared Memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[11] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[13] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. The fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[14] J. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[15] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[16] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[17] Manish Gupta. On Privatization of Variables for Data-Parallel Execution. In *Proceedings of the 11th International Parallel Processing Symposium*, Santa Barbara, CA, April 1997.

[18] 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.

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

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

[21] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[22] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

[23] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code (extended abstract). In *Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997. A full version of this paper was selected for publication in a special issue of the *International Journal of Parallel Programming*.

[24] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2), 1995.

[25] 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.

[26] A. Sawdey and M. O'Keefe. Program analysis of overlap area usage in self-similar parallel programs. In *Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science Series, Minneapolis, MN, August 1997. Springer-Verlag.

[27] Ajay Sethi. *Communication Generation for Data-Parallel Languages*. PhD thesis, Rice University, December 1996.

[28] Kees van Reeuwijk, Will Denissen, Henk Sips, and Edwin Paalvast. An implementation framework for hpf distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):897–914, September 1996.