Compiling Reductions in Data Parallel Programs for Distributed Memory Multiprocessors

Bo Lu

CRPC-TR97723-S
July 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997; Also available as Rice University M.S. Thesis

RICE UNIVERSITY

Compiling Reductions in Data Parallel Programs for Distributed Memory Multiprocessors

by

Bo Lu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. John Mellor-Crummey, Chairman
Faculty Fellow
Computer Science

Dr. Ken Kennedy
Noah Harding Professor
Computer Science

Dr. Linda Torczon
Faculty Fellow
Computer Science

Houston, Texas

July, 1997

Abstract

Compiling Reductions in Data Parallel Programs for Distributed Memory Multiprocessors

Bo Lu

Reduction recognition and optimization are crucial techniques in parallelizing compilers. They are used to detect the recurrences in a program and transform the originally sequential code into parallel code. Because of the expensive interprocessor communication cost, reduction recognition and efficient code generation become even more important for distributed-memory multiprocessors. As part of the dHPF parallel compiler project, we developed reduction recognition and parallel code generation strategies for distributed-memory multiprocessors.

Using combination of dependence analysis and pattern matching, our reduction recognition technique can detect a broad range of reduction operations in complex loop or control flow structures. It recognizes reductions into scalar variable as well as reductions into one or more elements of an array. To generate efficient code, we generalize dHPF's computation partitioning algorithm to assign computations for reduction statements and apply various techniques to reduce the number of collective communications. Furthermore, we "factor" some reduction statements into a group of reduction statements to better exploit data locality.

We have evaluated the effectiveness of our optimization on an IBM Scalable PowerParallel System SP2. The results indicate that without reduction optimization, reduction operations are sequentially executed and become an execution bottleneck; while with reduction support, we get good speedups and the time for reductions is almost negligible as compared to that for the non-optimized version. Compared to IBM's xlhpf compiler, dHPF can recognize a larger set of reduction operations and dHPF achieves 111% to 270% increase in efficiency for combined extreme value/location reductions. Our experiments show that factorization is an effective approach which reduces the reduction operation time by 50% or more where it is applicable.

Acknowledgments

I would like to thank my advisor Dr. John Mellor-Crummey for his guidance and valuable advice throughout this work. I would like to thank Dr. Ken Kennedy and Dr. Linda Torczon for serving on my committee in spite of their busy schedules. I also want to thank Dr. Vikram Adve, Ajay Sethi, Nenad Nedeljkovic, Mike Paleczny, Nat McIntosh, Collin McCurdy, Lei Zhou, Lisa Thomas, Monika Mevenkamp, and other dHPF group members for their knowledge and experience, and for being extremely helpful and reliable during the implementation of the compiler. My thanks to Collin McCurdy, for a thorough reading of my thesis and many insightful suggestions. I would like to thank Lennart Johnsson and University of Houston for providing their SP2 machines for our experiments. I am also grateful to Ellen Butler and Ivy Jorgensen for their assistance.

Contents

Tables

Illustrations

Chapter 1

Introduction

Developing software to obtain good performance on parallel computers has proven to be a difficult problem because of the programming complexity. Data parallel programming is an attractive programming paradigm where different processes perform the same operations on different portions of a data structure (such as an array). It reflects a simple execution model, which has a single-threaded control structure, global name space and loosely synchronous parallel execution. Data parallel programming scales well to large machines. Languages such as High Performance Fortran (HPF) provide mechanisms for specifying the mapping of data elements to processors along with aggregate operations that can be performed in parallel on data mapped to different processors. This high-level support aids construction of portable data-parallel programs. Optimizing compilers are required to exploit the parallelism inherent in data parallel programs to provide efficient implementations on parallel processors. Among many of the research issues arising in compiling data parallel programs, reduction handling is one with great importance.

A reduction is a commutative and associative operation which maps an array of n dimensions to an array of m dimensions, where $0 \leq m < n$*. Reductions occur explicitly in HPF programs when one uses FORTRAN 90 intrinsic functions. They also occur implicitly in programs when a location is updated on each loop iteration with the result of a commutative and associative operation applied to its previous contents. Reduction operations appear in many contexts such as kernels for matrix multiplication, image processing, computational geometry, and sorting among others.

When using a compilation model that discovers parallelism through dependence analysis, reductions are classified as inherently sequential operations, since the repeated updates to the same location are considered to be prohibitive to parallel execution of a loop. We can parallelize reduction computations safely by noticing that the ordering of the commutative updates does not need to be preserved. By

*an array of 0 dimension represents a scalar value.

applying program transformations, reductions can be optimized effectively for a variety of architectures. On parallel machines, reductions can be computed efficiently by having each processor compute a partial reduction result in parallel and then having processors synchronize to combine these partial results into the global reduction result. If we have a reduction of n elements on p processors, and the elements are evenly distributed, we can compute the result in time $O(n/p + log\ p)$ by parallelizing the reduction. Moreover, in compilers that don't support loop distribution, this difference is more substantial when a larger granularity of parallelism is exposed after "eliminating" the last loop-carried dependence.

HPF includes and extends many of the reduction intrinsic functions in FORTRAN 90. By using these operations, data parallel programs can be coded at a higher level, with potentially greater efficiency. However, HPF and FORTRAN 90 intrinsic functions are only able to express a limited set of reduction operations directly. For example, some of the *multi-element reductions*, which reduce the results to multiple elements in an array, such as $A(index) = A(index) + t$ and the *multi-dimensional reductions*, which carry out the reductions on more than one dimension, such as $S(i) = S(i) + A(i, j, k)$ are unimplementable using only a single invocation of an intrinsic function. Because not all programmers are aware of the occurrences of reductions or the existence of intrinsic functions, many reduction operations appear in an implicit form written by programmers. Calling to reduction intrinsics can always be transformed into an equivalent implicit form consisting of a loop nest that performs the reduction. Therefore, in our dHPF system, we have implemented a reduction recognition package that identifies reductions written in F77 sequential form and generates parallel code for them.

Reduction recognition and optimization are important techniques in an optimizing compiler for data parallel languages. We believe that with dependence analysis, pattern matching and flexible code generation support, we can provide efficient reduction support for distributed memory machines. Our goal is to develop a general reduction recognition method to recognize a broad range of reduction operations, along with an effective code generation strategy. Because of the high communication cost on distributed memory systems, we believe it is important to exploit data locality for generating efficient code.

Many of the previous approaches for detecting recurrences are principally based on pattern matching. The SUIF [9, 10] compiler uses a simple recognition strategy based on pattern matching the commonly occurring reductions. A unique strength

of SUIF is that it can recognize inter-procedural reductions. Polaris [4, 13] can recognize single address reductions† as well as histogram reductions‡ using an approach similar to ours. However, both SUIF and Polaris only generate code for reductions on shared memory systems, and haven't addressed the issues of data locality and communication optimization. The IBM HPF compiler [15] recognizes reductions into scalar variables. It generates code for the distributed-memory machines and has an optimization approach to coalesce and aggregate multiple reduction communications. However, it can not deal with multi-element reductions or multi-dimensional reductions. The need of doing forward substitution and reduction operand prefetching makes the reduction optimization impossible when the reduction operation is more closely intermixed with other computations.

Based on previous reduction recognition work in PFC [5], which was targeted for synthesizing FORTRAN 90 intrinsic reductions from their implicit form, in dHPF compiler we implemented reduction recognition for parallel reductions and generated reduction code for distributed-memory machines. Our work shows that efficient handling of reductions that takes data locality into account is crucial for obtaining high performance with HPF on distributed memory machines. On a 64-node SP2 machine, we experimentally evaluated our techniques and demonstrated significant performance improvements with our reduction optimizations.

The rest of this thesis is organized as follows. In Chapter 2 we give a detailed description of reduction recognition and briefly introduce some related work in this area. The implementation for reduction recognition and code generation are described in Chapter 3. The experimental evaluation are presented in Chapter 4. Chapter 5 presents conclusions of this work.

---

†single address reductions are the same as scalar reductions in our model.

‡histogram reductions are the same as array reductions in our model.

Chapter 2

Background

2.1 Reduction Optimizations

A reduction is an operation which maps an array of n dimensions to an array of m dimensions, where $m < n$ and $m \geq 0$. Many reductions occur implicitly in programs as statements that perform a repeated commutative and associative operation on the same element. The commutative and associative operations include SUM, PRODUCT, MIN/MAX, MINLOC/MAXLOC. As an example, the following loop and statement S_1 performs a sum reduction on the elements of A to produce a scalar S.

```
        DO k = 1, N
           ...
S1            S = S + A(k)
           ...
        ENDDO
```

Figure 2.1 Sum reduction example

Dependence analysis is a fundamental approach to uncover the potential parallelism in a program. There is a *data dependence* from statement S_1 to statement S_2 if both statements access the same memory location, at least one of them stores into it, and there is a feasible run-time execution path from S_1 to S_2 [11]. There are three types of data dependences: flow, anti, and output dependences. A *flow dependence* or *true dependence* exists if S_1 stores into a location that is later read by S_2. An *anti-dependence* exists if S_1 reads from a location into which S_2 later stores. An *output dependence* exists if both statements write into the same location. A dependence is said to be *loop-carried* if the instances of statements S_1 and S_2 that access the same memory location are in different loop iterations.

In the example, the statement $S = S + A(k)$ reads and writes S in one iteration, and also in the next iteration. So, there are loop-carried flow, anti, and output dependences on S between the instances of the statement in different loop iterations.

The Fundamental Theorem of Dependence states that any reordering transformation that preserves every dependence in a program preserves the meaning of the program [11]. Thus without knowledge otherwise, loop-carried dependences require the sequential execution of the loop iterations, and are prohibitive to the parallelization of the loop when the dependence is concerned.

However, a sequential order of operations need not always be preserved in the presence of dependences. Semantically, the code in figure 2.1 accumulates the sum of elements in array A. Sequential order does not need to be preserved between statement instances since the sum operation is commutative and associative.

After recognizing statement S_1 in Figure 2.1 as a reduction operation, the loop with statement S_1 can be computed more efficiently in several ways depending on the underlying hardware and software support, for example:

1. For machines with a pipelined addition operation, such as a vector machine, suppose it has a four-stage addition pipeline, the sum reduction can be decomposed into four separate sum reduction computations: [11]

   $S = 0.0$
   DO $k = 1, 4$
   $SUM(k) = 0.0$
   DO $I = k, N, 4$
   $SUM(k) = SUM(k) + A(I)$
   ENDDO
   $S = S + SUM(k)$
   ENDDO

   We can then apply loop distribution, loop interchange and vectorization to the above computation and get:

   $S = 0.0$
   DO $k = 1, 4$
   $SUM(k) = 0.0$

ENDDO
DO $I = 1, \frac{N}{4} \times 4, 4$
$SUM(1:4) = SUM(1:4) + A(I:I+3))$
ENDDO
$SUM(1 : N - \frac{N}{4} \times 4) = SUM(1 : N - \frac{N}{4} \times 4) + A(\frac{N}{4} \times 4 + 1 : N)$

DO $k = 1, 4$
$S = S + SUM(k)$
ENDDO

We can fill the four-stage addition pipeline with computations for $SUM(1) + A(I), SUM(2) + A(I+1), SUM(3) + A(I+2)$ and $SUM(4) + A(I+3)$, and the vector computation for $A(I+4)$ can begin immediately after the vector computation for A(I) has finished.

Once the four partial sums are available in SUM(1:4), the total sum can be computed by three floating point additions. Thus, sum reduction can be computed at nearly full vector speed.

2. **Parallel reduction** On a parallel machine, we can perform reductions in parallel with each processor computing a partial sum and then combining the results of individual processors at the end.

Chapter 3

Compiler Support for Reductions

Handling reductions in dHPF compiler proceeds in two phases. First we recognize the reduction patterns corresponding to the operations of SUM, PRODUCT, MIN, MAX, MINLOC and MAXLOC. Second, we generate parallel code to implement reductions for message passing systems that use the message passing interface (MPI) communication standard.

3.1 Reduction Recognition

In the reduction recognition phase, we use three program representations: a data dependence graph, single static assignment form (SSA) [11], and an abstract syntax tree (AST). A data dependence graph is used to identify the initial candidates for reduction statements; we use SSA to trace down the definitions of the right hand side variables and use AST to examine statement structures to discover the properties of the right hand side variables in a reduction statement.

There are two parts to the reduction recognition phase. First, for each assignment statement, we check if it is reducible; if it is reducible, we classify its reduction type. Then, we gather related reduction statements into reduction groups. For each group, we check if it is a reducible group (for example, all the statements in the group must have the same reduction type), and decide the levels that the reduction can be carried on.

For SUM or PRODUCT reductions, a reduction group is formed by reduction statements with the same left hand side (LHS) accumulator and same reduction operator. Not all statements need to be at the same loop level. For MIN/MAX reductions, a reduction group is formed by the assignment statement that record the extreme value along with any corresponding assignment statements that record the position of the extreme value. By forming a reduction group at reduction recognition time, we can use a single collective communication to get the reduction result in the code generation phase. This effectively eliminates the need of reduction coalescing implemented in

IBM HPF compiler [15], where they first generate separate reduction communication calls for each statement and then merge them together.

3.1.1 Single Statement Sum and Product Reductions

Consider a statement inside a possibly imperfect loop nest:

DO $i_1 = \ldots$
  DO $i_2 = \ldots$
    $\vdots$
    DO $i_n = \ldots$
      $A(\alpha_1, \alpha_2, ..., \alpha_n) = X \times A(\alpha_1, \alpha_2, ..., \alpha_n) + Y$ (3.1)
    ENDDO
    $\vdots$
  ENDDO
ENDDO

The statement (3.1) is a candidate for sum reduction if X is equal to 1 and Y is an expression that is constant with respect to (w.r.t.) $A(\alpha_1, \alpha_2, ..., \alpha_n)$[§] . Statement (3.1) is a candidate for product reduction, if $Y = 0$ and X is constant w.r.t. $A(\alpha_1, \alpha_2, ..., \alpha_n)$.

The reduction statement can be located anywhere in a loop nest, it can also be inside a control statement, such as:

DO $k = 1, upb$
  IF $(A(k).GT.0)$
    $T = T + A(k)$
  ELSE
    $T = T - A(k)$
  ENDIF
ENDDO

[§] x is said to be constant with respect to (w.r.t.) y when x is invariant to y for one or more loop levels.

Algorithm for the Single Statement Sum and Product Reductions

The outline of the algorithm for single statement sum and product reductions is shown in Figure 3.1. It takes a single assignment statement as an input and returns the appropriate reduction type. This single statement reduction algorithm is used by the reduction group algorithm described in Section 3.1.3 to check every assignment statement inside a loop.

3.1.2 Extreme Value Reductions

Using the AST, dependence graph and control flow graph data structures, we are also able to recognize MIN/MAX, MINLOC/MAXLOC reductions. For example:

```
DO I = 2, 100
   IF( C(I) .GT. MAX ) THEN
S1       MAX = C(I)
   ENDIF
ENDDO
```

S_1 is recognized as a MAX reduction.

```
DO I = 2, 100
   IF( C(I) .GT. MAX ) THEN
S1       MAX = C(I)
S2       MAXI = I
   ENDIF
ENDDO
```

S_1 S_2 are recognized as a group of statements for MAXLOC reduction.

Algorithm for Min/Max, MinLoc/MaxLoc Reduction

Figure 3.3 shows the algorithm to determine if statement S_1 is MIN/MAX(MINLOC/MAXLOC) reducible.

Input: a single assignment statement.
Output: a reduction type classification.

1. Use the dependence graph to check if there are loop-carried flow, anti and output dependences on the statement. If not, return RD_NO_REDUCE.
2. If the LHS is an array element, check if it is identical to each use of the variable on the rhs. If not, return RD_NO_REDUCE.
3. Use the algorithm shown in Figure 3.2 to check each term ($X \times A(\alpha_1, \alpha_2, ..., \alpha_n)$, Y in equation (3.1)) on the rhs. Follow the SSA edges and examine the AST structure to determine their relationships with the induction variable and LHS. This process is used to find out the corresponding expressions X, A, Y on the rhs. $X \times A(\alpha_1, \alpha_2, ..., \alpha_n)$ should be the only term that is not constant w.r.t. LHS, and Y represents the rest of the expressions on rhs. Based on the properties of X and Y, we can classify the reduction type of the statement.
4. Return the appropriate reduction type classification for this single statement. The five types are:
   (a) RD_IS_SUM: $X = 1, Y \neq 0$, and Y is not constant w.r.t. the induction variable.
   (b) RD_IS_PROD: $X \neq 1, Y = 0$, and X is not constant w.r.t. the induction variable.
   (c) RD_IS_ALGE_SUM: $X = 1, Y \neq 0$, and Y is constant w.r.t. the induction variable. We can compute the sum directly using a single statement:$A = A + Y \times iter$, where $iter$ is the number of iterations.
   (d) RD_IS_ALGE_PROD: $X \neq 1, Y = 0$, and X is constant w.r.t. the induction variable. We can compute the product directly using a single statement: $A = A \times X^{iter}$, where $iter$ is the number of iterations.
   (e) RD_NO_REDUCE: not reducible. This includes statements that failed the tests in steps 1 and 2, and also the statements that belong to none of the above categories.

Figure 3.1 Single statement sum and product reduction algorithm

We use a recursive process to check if X or Y is constant w.r.t. the induction variable and LHS. For example, if we want to check whether a variable t in expression Y is constant w.r.t. the induction variable i_n:

1. Check if i_n appears directly in t, if so, return non-constant.
2. If t is constant or it is not defined in loop with induction variable i_n, return constant.
3. Otherwise, we transitively follow the SSA edge to find the previous definition(s) for t.
4. For an assignment defining variable t, get the rhs expression.
5. Recursively examine each term on the rhs. If every term is constant w.r.t. i_n, the variable defined in this statement is constant w.r.t. i_n. Otherwise if one of the terms is not constant w.r.t. i_n, the variable defined in this statement is not constant w.r.t. i_n.

Figure 3.2 Algorithms to determine the properties of the rhs variables

Input: a single assignment statement S_1.
Output: reduction type classification and the reduction group formed by the related statements.

1. Make sure S_1 is immediately control dependent on an 'if' statement. Get the conditional expression for the 'if' statement.
2. For each statement immediately control dependent on the 'if', check if one of them is MIN/MAX reduction by looking for an assignment statement that records an extreme value.
3. If none of them is a possible MIN/MAX reduction statement, S_1 can not be MIN/MAX(MINLOC/MAXLOC) reducible. Otherwise
4. Check all the other statements that are immediately control dependent on the 'if' statement, make sure they are the possible MINLOC/MAXLOC reduction assignment statements recording the positions of the extreme value. If other statements exist, it is not a possible reduction.
5. Compare the MIN/MAX statements with the enclosing if statement. By looking at the operator in the conditional statement and the control dependence label of the branch the MIN/MAX statement is in, we can decide its reduction type. We can decide if it is MIN or MAX reduction, and in the case where location is computed, we can decide if it is MINLOC or MAXLOC reduction. Return the type and the relevant information.

Figure 3.3 Algorithm for extreme value reductions

3.1.3 Multistatement Reduction Groups for Sum and Product Reductions

A group of reducible statements with the same LHS accumulator and the same reduction operator can form a reduction group. For example, the following is a reduction group of SUM operations with the accumulator of T:

$T = T + y_1$
$T = T + y_2$
...
$T = T + y_n$

By forming the reduction group, we can store the local reduction result (a local accumulator for T in the example) in the same place for all the reduction statements in the group, and use a single collective communication operation to get the global result. The collective communication is effectively a barrier operation, which is quite expensive. In IBM HPF compiler [15], they first generate a separate communication event for each reduction operation, and then apply reduction coalescing and aggregation to merge some of the communication operations. By forming the reduction group and generating one communication event for each group, we effectively eliminate the need for communication coalescing in many cases.

In order to be a reduction group, the accumulator T can not be modified or referenced in other statements beyond the group in the same loop level. For example:

Do $i = 1, upb$
S_1 $T = T + y_1$
S_2 $T = T + y_2$
...
S_n $T = T + y_n$
$Q = T$
ENDDO

Since the value of T is needed inside the same loop, we can't use reduction optimization for statements S_1 to S_n. If the value of T is referenced or modified in other statements in an outer level, the final reduction result of T must be computed before the use or modification. So the place of references and modifications of T by other statements will decide how many levels the reduction can be carried out.

Reduction Levels

In our recognition package, we recognize the reduction groups at different levels in a program. We use the data dependence graph to detect modifications and references to the LHS to decide how many levels the reduction can be carried out. For example:

```
DO k = 1, upb1
S1      T = S + 9
G1      S = S + B(k)
        DO i = 1, upb2
G2          S = S + C(i)
            DO j = 1, upb3
G2              S = S + E(j)
            ENDDO
G2          S = S + D(i)
        ENDDO
ENDDO
```

The above loop nest contains 2 reduction groups G_1 and G_2 that contain the statements as labeled. Since S is referenced in S_1, which is not in any reduction group, we will need the value of S at statement S_1. Therefore, reduction optimization is not possible for statements in group G_1. On the other hand, since we can compute the local S out of order in loop i and j and provide the final reduction result in level k, before the use of S in statement S_1. Therefore, we say group G_2 is reducible in the inner two loops.

Algorithm For Building Reduction Groups

The process for finding reduction groups in a program is illustrated in Figure 3.4.

3.1.4 Reduction Idioms Recognized

As we show in Chapter 4, we can obtain good performance with reduction optimization, while reductions often become a bottleneck of programs without reduction optimization. So, it is very important to have a powerful reduction recognition pack-

Input: a Fortran 77 program.
Output: all the reduction groups in this program.

1. Build reduction groups for statements within the same loop.

   - For each assignment statement inside a loop, apply the algorithm shown in Figure 3.1 to get the reduction type for the statement.
   - If it is reducible and it is not a member of an existing group, build a group of reduction statements with the same accumulator, same reduction type within the same loop.

2. Merge reduction groups at different loop levels. If we have two reduction groups: G_1 in level m and G_2 in level n, where $m < n$. We can merge G_1 and G_2 if they have the same accumulator and reduction type and there is no reference or modification of the accumulator by other statements between levels n and m. Otherwise, if the reduction accumulator is referenced or modified in level k, where $m \leq k < n$, group G_1 in the outer loop is not reducible and we can not merge the two groups.

   Therefore, for each reduction group G_1 at level m:

   (a) Search all of the dependences of the LHS variable to find other statements that have referenced or modified it. If one such statement is in another reduction group G_2 which has the same accumulator and reduction operator and it is at some level $k \geq m$, we say G_1 and G_2 are compatible and mark G_2 as a candidate group to be merged with G_1 later. If the statement is not in another compatible reduction group, and it is at some level $k \geq m$, mark G_1 as "not reducible".

   (b) If G_1 is not marked as "not reducible", merge G_1 with all the compatible reduction groups at levels $k \geq m$, such as G_2 in the above example.

   (c) Decide the level which the reduction can be carried on based on the dependences we have checked in step (a).

Figure 3.4 Algorithm for building reduction groups in a program

age. In dHPF, the algorithms described earlier in this section can recognize a broad range of reduction operations:

- It can recognize scalar reductions which accumulate the results into a scalar variable, and multi-element array reductions which accumulate the reduction results into one or more elements in an array.
- It can detect reduction operations in any loop level, and can decide how many levels the reduction operation can be carried out.
- It can detect reduction operations inside any control flow statement, such as "`if` α `then` $s = s * a(i)$", and this reduction statement can form a reduction group with other statements outside the control statement.
- It can detect MIN/MAX and MINLOC/MAXLOC reduction written in different forms of `if` structures, and can recognize the MIN/MAX or MINLOC/MAXLOC for absolute values. We show such an example from the SPEC 92 Tomcatv benchmark program in the next chapter.
- It can recognize reductions which are closely intermixed with other computations. For example, it can use arrays or privatizable variables which are defined previously in the same loop while in most other compilers, they require the reduction elements to be prefetchable and they will isolate the reduction computation from other operations.

3.2 Code Generation

Compiling data parallel programs to distributed-memory machines can be separated into two major phases. The first phase determines how to decompose the computation and data across the processors. The goal is not just to parallelize the application, but also to minimize the communication overhead by minimizing the frequency of messages. In dHPF, data layouts for arrays are specified by HPF data layout directives in the user program. Based on data layouts, the compiler selects a computation partitioning, which is a map from each statement instance to a processor or multiple processors, that minimizes data movements. The second phase of the compilation is to generate the code so that each processor will execute its allotted computation and communication correctly and efficiently.

3.2.1 Computation Partitioning for Reduction Statements

Computation Partitioning Model in dHPF

dHPF supports a computation partitioning (CP) model [1] in which each statement in a loop may have a different partitioning. This differs from the CP model supported by SUIF [2] and Barua, Krans & Agarwal [3] which assigns a single CP to an entire loop iteration. This is also more general than the widely-used owner-computes rule [14]. Computation partitionings in dHPF allow each statement to have one or more computational "homes" that specify where instances of the statement will execute. For example, for a statement inside a loop nest with iteration space i, we can specify the computation partitioning (CP) to be the owner(s) of one data reference: ON_HOME $A_k(f_k(\vec{i}))$ or the owner(s) of several data references: $\cup_{k \in I}$ ON_HOME $A_k(f_k(\vec{i}))$, where I is a set of integers, and $\vec{i}$ is the vector of enclosing loop indices.

For programs without reductions, the CP selection algorithm in the dHPF compiler first assigns CPs for assignment statements, except for assignments to privatizable variables. For each statement, a CP is chosen to be ON_HOME of one of the references in the statement. Next, CP is propagated to control flow statements and assignments to privatizable variables. Control flow statements are assigned union of the CPs for the statements that are control dependent on them. Our current CP selection algorithm ensures that there is no communication for the values of privatizable variables by assigning the CP for the privatizable definition statement to be union of the CPs for all the uses of that privatizable variable.

Computation Partitioning for Reduction Related Statements

There are three steps in parallelizing a reduction operation. If we take sum reduction $S = S + A(i)$ as an example, in a preamble that is executed on every processor, we store the original value of S in a temporary variable T, and initialize S to be zero. In the reduction core, on each processor owning a part of array A involved in the reduction, we compute into S the partial sum of the local values. Finally, in a postamble, we accumulate the partial sums using a collective communication operation, and add back the original value saved in T to get the final sum. We assign a replicated CP to the preamble and postamble, that is, every processor initializes the partial sum and participates in adding up the final reduction value. The CP for the partial sum computation $S = S + A(i)$ would be ON_HOME $A(i)$ or ON_HOME of one of the references if there are several references on the right hand side.

3.2.2 Factorization and Data Locality

Suppose we have a reduction statement $S = S \oplus A_1(f_1(i)) \oplus A_2(f_2(i)) \ldots \oplus A_n(f_n(i))$, where there is more than one reference, namely $n > 1$. If not all of the references are distributed in the same way, we will have to read off-processor data to compute the partial reduction value no matter how we specify the CP for the above statement. For example, if we decide to compute the above statement ON_HOME $A_1(f_1(i))$, and $A_2(f_2(i))$ is not local to some of the processors who own $A_1(f_1(i))$, we have to do a non-local read to get the value of $A_2(f_2(i))$ to compute the sum ON_HOME $A_1(f_1(i))$.

We can eliminate the need to read off-processor data by splitting the above reduction statement into a sequence of statements:

$$S = S \oplus A_1(f_1(i)),\ S = S \oplus A_2(f_2(i)),\ \ldots,\ S = S \oplus A_n(f_n(i)) \quad (1)$$

This way, we can compute each reduction statement $S = S \oplus A_k(f_k(i))\ (1 \leq k \leq n)$ ON_HOME $A_k(f_k(i))$ and no communication is needed. The factorization process can be easily accommodated in our model. Each of the statements in (1) will be in the same reduction group and the final result can be accumulated in the postamble by a single collective communication operation.

3.2.3 Computation Partitioning for Reductions with Privatizables

If we have a reduction statement $S = S \oplus t$, where t is a privatizable variable defined in some previous statement in the loop, we can not assign a reasonable CP for this statement without looking at the CP for the definition of the variable t. Doing forward substitution is not always possible in the presence of control statements and it requires some extra preprocessing work, so we try to avoid it in our system.

Without transforming the original program, we extend the original CP algorithm in dHPF to assign CP for reductions on privatizables. It works to minimize communication by looking back at the definitions of the privatizables, and also maintains the invariant previously described for our CP selection algorithm that no communication will be needed for the values of privatizable variables. The extended algorithm is shown in Figure 3.5.

The extended CP selection algorithm extends the original algorithm to assign CPs for reduction statements, including reductions for privatizable variables. It is not

The CP algorithm assigns CP for statements one scope level at a time. For each level:
The statements that need CPs are classified into three sets:
atomNodeSet: includes all the assignments whose LHS is not privatizable.
propNodeSet: all the other statements, including the definition for the privatizables, the loop nodes, and if statements. We need to do propagations to get CPs on these statements.
unknownNodeSet: The statements in atomNodeSet, which do not have any CP choices. The set is initially empty, we will fill it up when going through statements in atomNodeSet at the following step 1.

The algorithm works in three steps to assign CP for statements at each level:

1. For each statements in atomNodeSet, try to assign CP for it:
   - Compute the CP choices for the statement. If no choice for the statement, put it to unknownNodeSet.
   - Iterate through the choice map, using a cost estimator to choose the CPs for the statements in set: atomNodeSet-unknownNodeSet.
2. Propagate CPs from the set of atomNodeSet-unknownNodeSet to propNodeSet based on two rules:
   (a) The CP for the definition of the privatizable is the union of the CPs for the uses of the privatizable. So, we can make sure no communication is needed for the privatizables.
   (b) The CP for the loop node or the guard statement is the union of the CPs for the nodes that are control dependent on them.
3. Compute CPs for statements in the unknownNodeSet.
   For each statement in unknownNodeSet, we call function ComputeCPForNode() to get the CP.

   ComputeCPForNode(node):
   - If CP is already computed, return;
   - If there are arrays on the RHS, choose the CP to be on-home of one of them, which will cause the minimum amount of non-local data read for the right hand arrays;
   - If the above two fail, then the statement is in the form of $t = t_1 + t_2 + ... + t_n$. Find the definition of the privatizables $t_1, t_2, ..., t_n$. Choose a "majority CP" from the CPs we get from ComputeCPForNode(t_1), ComputeCPForNode(t_2), ... ComputeCPForNode(t_n).
4. Do a propagation again, this time. we propagate CP from the whole set of atomNodeSet to propNodeSet, to make the CP satisfy the two rules in 2.(a), 2.(b)

Figure 3.5 CP propagation algorithm for reductions with privatizables

globally optimal[¶], but it will work well for the typical programs while still comforming to the CP propagation invariants.

3.2.4 Code Generation for Distributed Memory Machines

As described in the previous section, there are three steps in a parallel reduction: preamble, local reduction computation and postamble. Implementing them is architecture specific. In the current dHPF compiler, we generate message-passing code for distributed-memory machines. The output is the F77 code with MPI communication primitives.

Code Generation for Sum/Product Reductions

The implementation for scalar reductions is straightforward. Since each scalar is replicated on every processor, we can use it to store the local reduction value. In the preamble, we store the original value of the scalar to a temporary variable and initialize the local reduction value to be Φ[‖]. In the reduction core, we compute a partial reduction result on each processor using contributions from local data. In the postamble, we call MPI_ALLREDUCE to compute the global reduction value by combining all of the partial reduction results from different processors, and to return the global reduction value to all the processors. Finally, each processor combines the global reduction value with the original value stored in the temporary variable to get the final result.

For a multi-element array reduction, the array is distributed among the processors. In the buffer space of each processor, we allocate a contiguous region for the whole array to store the partial reduction results. In the preamble, we initialize them to Φ. In the reduction core, we compute the partial reduction results in the buffer using contributions from local data. In the postamble, we call MPI_ALLREDUCE to propagate the global reduction results to every processor. Finally, for every element in the array, the owner(s) of that element is(are) responsible for combining the original value with the global reduction result, and storing the final result. Since the partial reduction values for all the array elements are stored in a contiguous space, we only need a single MPI_ALLREDUCE call to accumulate the global reduction result for all

[¶]If we have privatizable variable assignment statement$S : t = A(i) + t_1$, we will not look at the definition of t_1 to decide the CP for S, which may not be optimal.

[‖]Φ is 0 for sum reduction and 1 for product reduction

the elements in an array. This is much more efficient than calling MPI_ALLREDUCE separately for each element in the array, since each MPI_ALLREDUCE call is effectively a barrier involving a tight synchronization between processors.

Code Generation for Min/Max, MinLoc/MaxLoc Reductions

The code generation for a MIN/MAX reduction is similar to code generation for SUM/PRODUCT scalar reductions, except that the initial value will participate in the partial MIN/MAX selections, and the global reduction result from MPI_ALLREDUCE will be the final result.

The semantics of MPI_ALLREDUCE for MPI_MINLOC/MPI_MAXLOC is to compute a global minimum/maximum and also the index attached to that value. The operation that defines MPI_MAXLOC is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ min(i,j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

If we only need to get one index in the MINLOC/MAXLOC reduction and if the minimum index is desired when there are multiple locations possessing the same extreme value, we have each processor compute a pair of (local extreme value, index of local extreme value), and apply reduction operation MPI_MINLOC/MPI_MAXLOC to get the global min/max result along with the minimum index among all the indices for the global min/max values.

If multiple indices are needed for the final result, such as the following loop nest:

```
DO K=1,N
   DO J=1,N
      DO I=1,N
         IF( C(I, J, K) .GT. MAX ) THEN
S1             MAX = C(I, J, K)
```

```
S2              MAXI = I
S3              MAXJ = J
S4              MAXK = K
          ENDIF
       ENDDO
   ENDDO
ENDDO
```

which computes the global maximum value and its location coordinates, a straightforward way to parallelize this computation is to form a pair of (local max, processor rank) on each processor. After applying MPI_MAXLOC, we will get the global MAX result along with the lowest rank of the processor(s) that owns(own) an element with the maximum value. Then, that processor can broadcast the values of MAXI, MAXJ, MAXK to other processors and every processor can get the final result. It takes 2 collective communication operations: a reduction call followed by a broadcast.

We can eliminate the need for the broadcast by having each processor "encode" the coordinates of its local extreme value into a linearized coordinate space:
codeForCoordinates=
$((MAXK - LB_k) * (UB_j - LB_j) + (MAXJ - LB_j)) * (UB_i - LB_i) + (MAXI - LB_i)$

and form the pair (local max, *codeForCoordinates*) on each processor. After applying MPI_MAXLOC, we will get the global MAX result along with the encoded coordinates for the first global MAX value in a sequence of processors. Then we can decode the value to get the final result. Compared to the previous approach, this one only takes one collective communication.

The above encoding strategy gets the maximum value and the location for the maximum value. If there are more than one elements that have the same maximum value, it gets the smallest coordinates (I,J,K) in dictionary order. However, there are more complicated cases where we have different operators, such as operator "GE" instead of "GT", or we may iterate through one or all of the loops in a descending order. Below, we describe an encoding strategy that accommodates each of these variations and enables the reduction program to produce the same coordinates as we would get in a sequential execution order.

For example, in the following program:

```
DO J=1, N
   DO I=N,1,-1
      IF( C(I, J) .GE. MAX ) THEN
         MAX = C(I, J)
         MAXI = I
         MAXJ = J
      ENDIF
   ENDDO
ENDDO
```

the operator "GE" decides that we should get the coordinates appearing last. Since loop J iterates in an ascending order, and loop I iterates in a descending order, we should get the coordinates with the biggest J and the smallest I when J is equal. Therefore, we can encode the coordinates this way:

$$codeForCoordinates = (UB_j - MAXJ) * (UB_i - LB_i) + (MAXI - LB_i)$$

and get the same location result as that in a sequential execution order.

In general, if we have a loop nest of level n, and UB_m, LB_m are the upper bound and lower bound values for the induction variable at loop level m ($0 \leq m \leq n$, and $m = 1$ for the innermost loop), then *codeForCoordinates*=:

$$(...(C_n * (UB_{n-1} - LB_{n-1}) + C_{n-1}) * (UB_{n-2} - LB_{n-2}) + ...)(UB_1 - LB_1) + C_1$$

The coefficient C_m is determined by the type of the operator and the traversal order at loop level m. The pseudo code in Figure 3.6 describes the iterative procedure to produce *codeForCoordinates*.

The encoding strategy can also be used in some other places. For example, in tomcatv reductions (Figure 4.4), we want to get the value and location of the element that has the maximum absolute value. We generate a pair of (local absolute max, $2\times$*codeForCoordinates*+*sign*) on each processor, where "local absolute max" is the maximum absolute value on each processor. *codeForCoordinates* is generated the same way as described before, and *sign* represents the actual sign of the element that contributes the maximum absolute value on each processor where "0" stands for positive number and "1" stands for negative number. We can therefore decode "$2\times$*codeForCoordinates*+*sign*" to get the *sign* and the *codeForCoordinates*. By combining the *sign* with the global "absolute max", we can get the correct value of the

Input: an extreme value reduction.
Output: *codeForCoordinates* for the reduction.

```
Procedure Encode()
begin
   Boolean FIRST := (operator∉{GE,LE} ? true , false);
   for loop nest at level m, m from n to 1
   begin
       Boolean ASCENDING := (loop at level m traverses iteration in an ascending order ? true , false);
       Boolean SMALLEST := (ASCENDING && FIRST) || (!ASCENDING && !FIRST);
       if(SMALLEST)
           Coefficient := gencode("-", "Loc_m", "LB_m");
       else
           Coefficient := gencode("-", "UB_m", "Loc_m");
       if(m = n)
           Code := Coefficient;
       else
       begin
           Code := gencode("*", Code, gencode("-", "UB_m", "LB_m"));
           Code := gencode("+", Code, Coefficient);
       end
   end
   return Code;
end
```

Figure 3.6 Pseudo-code to encode coordinates for extreme value reductions

element that contributes the maximum absolute value. By decoding the *codeForCoordinates*, we can get the location of the maximum element. Using the encoding strategy, we can invoke a single MPI_ALLREDUCE call to get the extreme value along with the location of the value. The generated code for the tomcatv reductions is shown in Figure 4.5.

Chapter 4

Evaluation

We have evaluated the effectiveness of our reduction optimization strategy on an 64-processor IBM Scalable PowerParallel System SP2. We use the message passing interface (MPI) communication standard and the SP2 US communication subsystem library implementation which supports user-space messaging and requires the user to have dedicated use of a high-performance switch adapter.

IBM xlhpf is a High Performance Fortran compiler for machines running the AIX operating system such as the IBM SP2. It's one of the few compilers that supports reduction optimization for distributed-memory machines. We compared the available xlhpf 1.1.0.0 compiler with dHPF for the effectiveness of the reduction recognition and optimization. We also evaluated the speedups achieved by reduction support and measured the performance gain of the factorization strategy in dHPF.

4.1 Reduction Performance in dHPF

Erlebacher is an 600 line, ten procedure benchmark program from ICASE that performs three-dimensional tridiagonal solves. It includes a number of fully parallel computations, interleaved with multi-dimensional reductions and computational wavefronts in all three dimensions that are caused by forward and backward substitutions. We tested our dHPF compiler on a multi-dimensional reduction kernel extracted from erlebacher as shown in Figure 4.1.

Without reduction support, S_1 will be computed ON_HOME tot(i,j), which is replicated on all processors. Figures A-1, A-2 and A-3 in the Appendix show the code generated by dHPF without reduction support. There are `send`s and `recv`s between every two processors to provide each processor with a full local copy of array `duz`, then every processor will do the computation and get the final results.

On the other hand, applying the dHPF compiler with reduction support enabled, we get the generated code as shown in Figure 4.2. The figure shows only the reduction preamble, reduction core and postamble. Code preceding and following it has been

```
      program erlebacherKernel
      integer n
      parameter (n=64
      real duz(n,n,n), tot(n,n), d(n)
      parameter (n$proc = 4)
CHPF$ processors  p(n$proc)
CHPF$ template    dtempl(n,n,n)
CHPF$ template    vtempl(n$proc)
CHPF$ distribute dtempl(*,*,block) onto p
CHPF$ distribute vtempl(block) onto p
CHPF$ align duz(i,j,k) with dtempl(i,j,k)
CHPF$ align d(*) with vtempl(*)
CHPF$ align tot(*,*) with vtempl(*)
      do 30 j=1,n
         do 30 i=1,n
            tot(i,j) = 0.
 30      continue
      do 40 k=1,n-1
         do 40 j=1,n
            do 40 i=1,n
S1
               tot(i,j) = tot(i,j) + d(k)*duz(i,j,k)
 40         continue
      end
```

Figure 4.1 Reduction kernels in erlebacher

elided. In the preamble(lines 8 to 21), we allocate a temporary for array `tot` on the heap on each processor(lines 8 to 14), and initialize all the values to be zero(lines 15 to 21). In the reduction core(lines 22 to 30), we compute the partial reduction results into the temporary. Finally, in the postamble(lines 31 to 40), we invoke a MPI_ALLREDUCE to get the global reduction results for all the elements, get the final value and store them back into `tot`(lines 35 to 40).

We compare the execution time for the above programs generated by dHPF compiler with and without reduction optimization. Table 4.1 shows the timing results in seconds, and Figure 4.3 shows the speedups. The speedup is computed by t_s/t_p, where t_s is the sequential execution time and t_p is the parallel execution time.

Without reduction support, we send and receive $O(p \times (p-1))$ messages of size $\frac{n^3}{p}$, if we assume there are p processors. The frequency and total volume of communication will increase as the number of processors increases. In our current implementation,

```
      ...
      do j = 1, 64
        do i = 1, 64
          tot(i, j) = 0.
        enddo
      enddo

8     counter_tot_6 = 4096
      call hpf_buffer_alloc(counter_tot_6 * 4, send_buf_tot_6)
      call hpf_ptr_to_index(hpf_heap, send_buf_tot_6, 4, send_buf_tot_
     *6_index)
      call hpf_buffer_alloc(counter_tot_6 * 4, recv_buf_tot_6)
      call hpf_ptr_to_index(hpf_heap, recv_buf_tot_6, 4, recv_buf_tot_
14   *6_index)
15    counter_tot_6 = 0
      do iv_1 = 1, n
        do iv_0 = 1, n
          hpf_heap_real(send_buf_tot_6_index + counter_tot_6) = 0
          counter_tot_6 = counter_tot_6 + 1
        enddo
21    enddo
22    k_end__dhpf = min(16 * p_myid1 + 16, 63)
      do k = 16 * p_myid1 + 1, k_end__dhpf
          do i = 1, 64
            hpf_heap_real(send_buf_tot_6_index + i * 64 + j - 65) = hp
     *f_heap_real(send_buf_tot_6_index + i * 64 + j - 65) + d(k) * duz(i
     *, j, k - p_myid1 * 16 - 1)
          enddo
        enddo
30    enddo
31    call mpi_allreduce(hpf_heap_real(send_buf_tot_6_index), hpf_heap
     *_real(recv_buf_tot_6_index), counter_tot_6, mpi_real, mpi_sum, mpi
     *_comm_world, ierr)
      call hpf_buffer_free(send_buf_tot_6)
35    do iv_1 = 1, 64
        do iv_0 = 1, 64
          tot(iv_0, iv_1) = tot(iv_0, iv_1) + hpf_heap_real(recv_buf_t
     *ot_6_index + iv_0 * 64 + iv_1 - 65)
        enddo
40    enddo
      call hpf_buffer_free(recv_buf_tot_6)
      ...
```

Figure 4.2 Code generated for erlebacher reduction kernel with reduction support

nprocs	1	2	4	8	16	32
without reduction	0.04	6.29	25.25	11.60	22.72	27.80
with reduction	0.0405	0.0285	0.0201	0.0275	0.1671	0.6263

Table 4.1 Execution time for erlebacher reduction kernel with and without reduction optimization

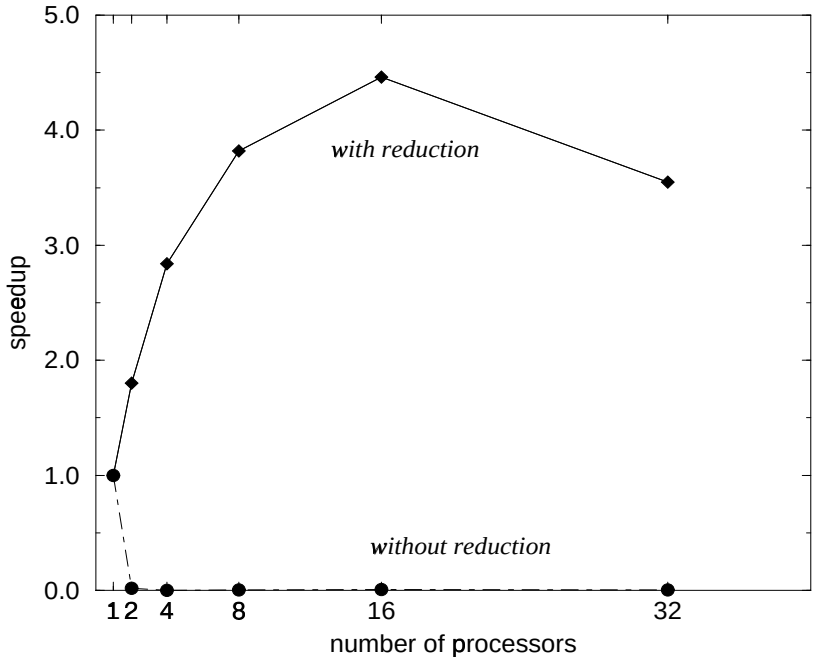

Figure 4.3 Speedups for erlebacher redcution kernel(N=514, 100 iterations)

since the communication is among multiple processors, we unpack the data from the system buffer and put each array element value into a hash table accessed by element index. When the number of processors is small and there is a large amount of data, we spend a large portion of time inserting data into a hash table, which involves many rehashing operations as the table is resized dynamically to accomodate all of the data. In the results, we can see a peak time for 4 processors, and a stable increase for 8 processors and more. We can't get any real speedup and reduction computation becomes a program bottleneck.

The time for programs with reduction support is almost negligible compared to programs without reduction support. The computation time is $O(\frac{n^3}{p})$, while the reduction commmunication time is O($n^2 \times$ log p). So, we see a time decrease to the lowest point at 4 processors and a gradual increase afterwards. In spite of the expensive collective communication cost, we still can get real speedups up to large number of processors.

We apply xlhpf on the same erlebacher reduction kernel program. xlhpf fails to recognize the multi-dimensional reduction operation, and generates the code similar to the one in dHPF without reduction support where the reduction operation is computed on every processor.

4.2 Comparison with Reductions in xlhpf

Compared to the dHPF compiler, xlhpf recognizes a smaller set of reductions. It does not recognize multi-dimensional reductions and or reduction operations in which privatizable variables appear on the right hand side, such as: $t = A(i), S = S + t$. It only recognizes MIN/MAX, MINLOC/MAXLOC reductions written in FORTRAN 90 intrinsic functions while dHPF can recognize the reduction written in implicit sequential form with different forms of control flow structures. We compare the two compilers using a reduction kernel from tomcatv program from the SPEC92 floating-point benchmark suite. It has two groups of MAXVAL, MAXLOC reduction operations. The code in Figure 4.4 shows the reduction kernels in the original FORTRAN program for 4 processors, and we use it as the input to dHPF compiler:

dHPF recognizes the above as two groups of reduction operations, and generates two MPI_ALLREDUCE calls to get the maximum values along with the locations. The code is listed in Figure 4.5.

```
...
CHPF$ PROCESSORS P(4)
CHPF$ ALIGN RX(I,J) with T(I,J)
CHPF$ ALIGN RY(I,J) with T(I,J)
CHPF$ DISTRIBUTE T(BLOCK,*) ONTO P
      I1P  = 2
      J1P  = 2
      I2M  = N-1
      J2M  = N-1
      M = J2M - J1P + 1
      RXM = 0
      RYM = 0
      DO    w = 1, iterations
      DO    270    J = 1,M
      DO    270    I = I1P,I2M
         IF(ABS(RX(I,J)).GT.ABS(RXM))  then
            RXM  = RX(I,J)
            IRXM = I
            JRXM = J
         endif
         IF(ABS(RY(I,J)).GT.ABS(RYM))  then
            RYM  = RY(I,J)
            IRYM = I
            JRYM = J
         endif
 270  CONTINUE
C     use of RXM, IRXM, JRXM,... here
      ENDDO
      END
```

Figure 4.4 Tomcatv reduction kernels

xlhpf compiler does not recognize the tomcatv reductions in their implicit form. For a performance comparison, we recoded the original reduction kernel to use the FORTRAN 90 intrinsic functions, forms that xlhpf can recognize (Figure 4.6 and Figure 4.7).

There is a slight difference between the two versions in Figure 4.6 and 4.7. Version 1 gets the maximum absolute value while version 2 gets the value of the element who has the maximum absolute value as in the original program.

In xlhpf, for program in version 1, it generates two separate loop iterations to get the local maxval and maxloc, then it invokes _xlhpf_reduce_maxval, _xlhpf_reduce_maxloc calls to get the global result. For program in version 2, it invokes _xlhpf_reduce_maxloc

```
      ...
      do w=1,100
       do j = 1, 512
         i_start__dhpf = max(129 * p_myid1 + 1, 2)
         i_end__dhpf = min(129 * p_myid1 + 129, 513)
         do i = i_start__dhpf, i_end__dhpf
           abs_result_0__dhpf = abs(rx(i - p_myid1 * 129 - 1, j))
           abs_result_1__dhpf = abs(rxm_tmp(1))
           if (abs_result_0__dhpf .gt. abs_result_1__dhpf) then
             rxm_tmp(1) = rx(i - p_myid1 * 129 - 1, j)
             irxm = i
             jrxm = j
           endif
C  same code here for 'y' to get the local extreme value
         enddo
       enddo
       rxm_tmp(2) = (jrxm - 1) * 513 - 1 + irxm
       rxm_tmp(2) = 2 * rxm_tmp(2)
       rxm_tmp_rxm_tmp0__dhpf = rxm_tmp(1)
       if (rxm_tmp_rxm_tmp0__dhpf .lt. 0) then
         rxm_tmp(2) = rxm_tmp(2) + 1
       endif
       rxm_tmp(1) = abs(rxm_tmp(1))
       call mpi_allreduce(rxm_tmp, rxm_res, 1, mpi_2double_precision, m
     *pi_maxloc, mpi_comm_world, ierr)
       rxm = rxm_res(1)
       codeForCoordinates = rxm_res(2) / 2
       sign = rxm_res(2) - codeForCoordinates * 2
       if (sign .gt. 0) then
         rxm = -rxm
       endif
       jrxm = codeForCoordinates / 513 + 1
       irxm = codeForCoordinates - (jrxm - 1) * 513 + 1
C   same code here for 'y' to get the global extreme value
      enddo
      ...
```

Figure 4.5 Code generated for tomcatv reduction kernel by dHPF compiler

```
...
INTEGER, DIMENSION (2)                :: MAX_LOCX
INTEGER, DIMENSION (2)                :: MAX_LOCY
DO    w = 1, iterations
  RXM = MAXVAL(ABS(RX(1:M, I1P:I2M)))
  MAX_LOCX = MAXLOC(ABS(RX(1:M, I1P:I2M)))
  IRXM = MAX_LOCX(1)
  JRXM = MAX_LOCX(2)+I1P-1

  RYM = MAXVAL(ABS(RY(1:M, I1P:I2M)))
  MAX_LOCY = MAXLOC(ABS(RY(1:M, I1P:I2M)))
  IRYM = MAX_LOCY(1)
  JRYM = MAX_LOCY(2)+I1P-1
ENDDO
END
```

Figure 4.6 Recoded tomcatv reduction kernels, Version 1

to get the location value and the processor who owns the element of RX(IRXM, JRXM) will broadcast the extreme value to all the other processors. Therefore, xlhpf will generate 4 collective communication calls for either one of them, instead of 2 in our compiler. We use the program in Figure 4.6 as the source program for xlhpf, and Figure A-4 through A-6 list the code generated by the compiler.

We compare the execution time for programs generated by xlhpf and dHPF on an IBM SP2. Table 4.2 and Figure 4.8 show the timing results and speedups while table 4.3 compares the efficiencies. The efficiency is computed by s/p, where p is the number of processors and s is the speedup on p processors.

nprocs	1	2	4	8	16	32
xlhpf	4.21	7.79	4.58	2.46	1.41	1.47
dHPF	4.21	3.37	2.08	1.17	0.66	0.40

Table 4.2 Execution time of tomcatv reduction kernels for xlhpf and dHPF

The time for dHPF is less than half of the time for xlhpf in almost all the cases. As discussed in the end of section 4.1, the computation time is $O(\frac{N}{p})$ while the communication time is O(log p). We can see a time decrease to the lowest point

```
...
INTEGER, DIMENSION (2)                :: MAX_LOCX
INTEGER, DIMENSION (2)                :: MAX_LOCY
DO    w = 1, iterations
  MAX_LOCX = MAXLOC(ABS(RX(1:M, I1P:I2M)))
  IRXM = MAX_LOCX(1)
  JRXM = MAX_LOCX(2)+I1P-1
  RXM = RX(IRXM, JRXM)

  MAX_LOCY = MAXLOC(ABS(RY(1:M, I1P:I2M)))
  IRYM = MAX_LOCY(1)
  JRYM = MAX_LOCY(2)+I1P-1
  RYM = RY(IRYM, JRYM)
ENDDO
END
```

Figure 4.7 Recoded tomcatv reduction kernels, Version 2

nprocs	1	2	4	8	16	32
e_1(xlhpf)	1	0.270	0.230	0.210	0.190	0.089
e_2(dHPF)	1	0.620	0.505	0.451	0.401	0.329
$e_2 - e_1$		0 0.350	0.275	0.241	0.211	0.240

Table 4.3 Efficiency of tomcatv reduction kernels for xlhpf and dHPF

at 16 processors and a gradual increase afterwards for xlhpf compiler and a time decrease up to 32 processors for dHPF compiler. We also get better speedups for dHPF compiler. The efficiency for dHPF compiler ranges from 0.329 to 0.620, while the efficiency for xlhpf ranges from 0.089 to 0.270, with an 111% to 270% increase in efficiency for dHPF compiler.

4.3 Performance Impact of Reduction Factorization

As discussed in 3.2.2, we use factorization to divide one reduction statement into a group of reduction statements to exploit the data locality. We test the reduction with and without factorization for the program in Figure 4.9.

There are more than one reduction components on the right hand side: a(n-j+1) and b(j). The two array elements are distributed in a different way. By doing reduc-

tion factorization, we can split the reduction statement into a group of two statements: s=s+t3 and s=s+b(j). s=s+t3 will get the CP of ON_HOME a(n-j+1) as specified by the computation partitioning algorithm for reductions with privatizable variables, and s=s+b(j) will get the CP of ON_HOME b(j). All the assignment statements are computed locally without the need for any communication. The generated code is shown in Figure 4.11:

On the other hand, if we don't have factorization support, statement s = s + t3 + b(j) will get the CP of ON_HOME b(j). t3=a(n-j+1) will get the same CP of ON_HOME b(j) by the CP propagation for privatizable variables. In this case, we need pairwise communications to get the non-local data of a(n-j+1) to compute statement t3=a(n-j+1) (Figure 4.10).

If there are p processors, the computation time will be $O(\frac{n}{p})$, the time for the global reduction will be O(log p). If there is no factorization support, we will need to send and receive non-local data between processors, and it will take an extra time of $O(\frac{n}{p})$. We measure the execution time of the above program for 4 processors with different values of n. We measure the code generated by the dHPF compiler, with and without factorization support enabled. In table 4.4, we list the execution time for programs with and without factorization support and also show the extra communication time for the one without factorization support.

Figure 4.12 shows the extra communication overhead incurred for not having factorization support for the reduction. In the programs without factorization support, the time for the non-local data communication takes about half of the total execution time.

We apply xlhpf compiler on the above program. It fails to recognize the reduction in the original form . We recode the source program by adding $s = 0$ inside loop $do\ w = 1, 100$. xlhpf recognizes the reduction in the transformed program. However, it generates code similar to our code without reduction factorization.

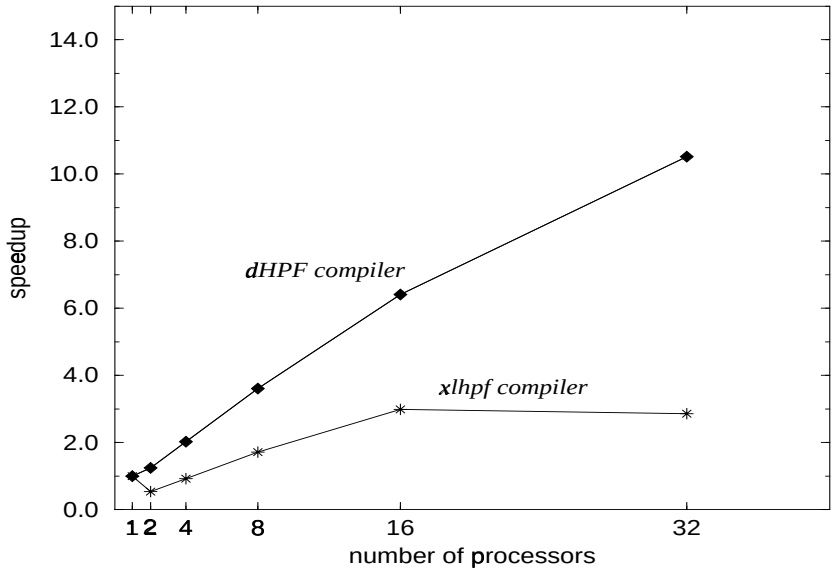

Figure 4.8 Speedups for tomcatv reduction kernels

n	4096	4096*4	4096*8	4096*16	4096*32
t_1(with factorization)	0.0379	0.0843	0.1489	0.2765	0.5250
t_2(without factorization)	0.0669	0.1883	0.3285	0.5933	1.0364
$\Delta = t_2 - t_1$	0.0290	0.1040	0.1796	0.3168	0.5114
$p = \frac{\Delta}{t_1}$	0.765	1.219	1.205	1.146	0.974

Table 4.4 Comparison of programs with and without factorization support

```
        program facTest
        integer n, w
        parameter (n=4096)
        real a(n), b(n)
        real s, t3
CHPF$   processors p(64)
CHPF$   template t(n)
CHPF$   align a(i) with t(i)
CHPF$   align b(i) with t(i)

CHPF$   distribute t(block) onto p
        s=0

        do w=1,100
           do j = 1, n
             a(j) = j
             b(j) = j
           enddo

           do j = 1, n
             t3 = a(n-j+1)
             s = s + t3 + b(j)
           enddo
        enddo
        end
```

Figure 4.9 Source program for factorization test

```
   s = 0
   do w=1,100
   s_tmp = s
   s = 0
   if (p_myid1 .le. 1) then
     do j = 1024 * p_myid1 + 1, 1024 * p_myid1 + 1024
       s = s + b(j - p_myid1 * 1024 - 1)
     enddo
   endif
   do j = -(1024 * p_myid1) + 3073, -(1024 * p_myid1) + 4096
     t3 = a(n - j - p_myid1 * 1024)
     s = s + t3
   enddo
   if (p_myid1 .ge. 2) then
     do j = 1024 * p_myid1 + 1, 1024 * p_myid1 + 1024
       s = s + b(j - p_myid1 * 1024 - 1)
     enddo
   endif
   call mpi_allreduce(s, s_res, 1, mpi_real, mpi_sum, mpi_comm_worl
*d, ierr)
   s = s_res + s_tmp
   enddo
   end
```

Figure 4.10 Code generated by dHPF with factorization support

```
        s = 0
        do w=1,100
        p_q1 = -p_myid1 + 3
C       ***** MPI SEND STMT FOR NONLOCAL READ *****
        counter_a_4 = 0
        lb = -p_q1 * 1024 - p_myid1 * 1024 + 3072
        ub = min(4096, -p_q1 * 1024 + 4096) - p_myid1 * 1024 - 1
        counter_a_4 = ub - lb + 1
        if (counter_a_4 .gt. 0) then
          call mpi_bsend(a(-p_myid1 * 1024 - p_q1 * 1024 + 3072), counte
     *r_a_4, mpi_real, p_q1, 1, p_cmap, ierr)
        endif
        s_tmp = s
        s = 0
        p_q1 = -p_myid1 + 3
C       --< Loop Counters >--
        counter_a_4 = 0
        counter_a_4 = counter_a_4 + 1024 * p_q1 + 1024 - (1024 * p_q1 +
     *1) + 1
        call hpf_buffer_alloc(counter_a_4 * 4, recv_buf_a_4)
        call hpf_ptr_to_index(hpf_heap, recv_buf_a_4, 4, recv_buf_a_4_in
     *dex)
        if (counter_a_4 .gt. 0) then
          call mpi_recv(hpf_heap_real(recv_buf_a_4_index), counter_a_4,
     *mpi_real, p_q1, 1, p_cmap, status, ierr)
        endif
        do j = 1024 * p_myid1 + 1, 1024 * p_myid1 + 1024
          if (p_myid1 .eq. (n - j) / 1024) then
            lnltmp0 = a(n - j - p_myid1 * 1024)
          else
            lnltmp0 = hpf_heap_real(recv_buf_a_4_index + n - j - p_q1 *
     *1024)
          endif
          t3 = lnltmp0
          s = s + t3 + b(j - p_myid1 * 1024 - 1)
        enddo
        call mpi_allreduce(s, s_res, 1, mpi_real, mpi_sum, mpi_comm_worl
     *d, ierr)
        s = s_res + s_tmp
        enddo
```

Figure 4.11 Code generated by dHPF without factorization support

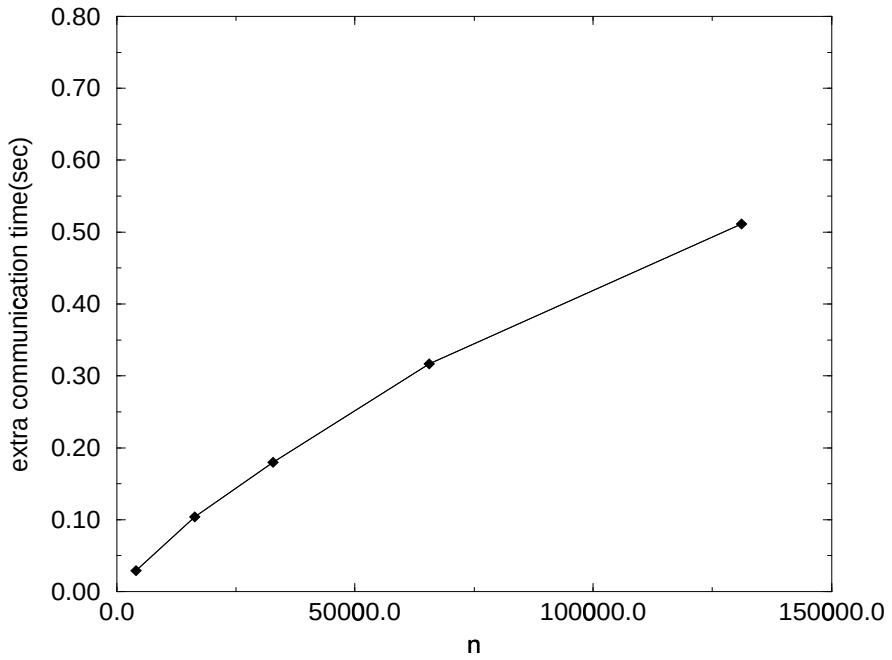

Figure 4.12 Time overhead for reductions without factorization support(p=4)

Chapter 5

Related Work

Our reduction recognition work is based on the recognition techniques developed by Darnell in PFC [5]. We use a similar pattern matching approach, and incorporate dependence analysis. The previous technique recognizes the reductions and transforms it into FORTRAN intrinsic function calls, so it can only handle a small set of reductions and requires the reduction operations be isolated from other computations. Techniques implemented by Darnell recognize sum and product reductions within the same loop level, but haven't considered the extreme value reductions.

Fisher and Ghuloum [8, 7] have a unique and powerful technique for recognizing reductions and scan operations. They model complex scan and reduction operations as function applications and extract parallelism by exploiting the associativity of functional compositions. However, their recognizer implicitly assumes that the recurrence loops are separated from other code. Their code generation model is simple in that they initialize the template variables with array values prior to composition without considering the locality of the array elements. Their techniques are especially useful for parallelizing scan operations which are not handled by most compilers. However, reductions, as a special case, are not handled as efficiently as in our compiler.

The SUIF compiler [9, 10] supports inter-procedural reduction recognition. It recognizes summation, multiplication, minimum and maximum reduction operations inter-procedurally. However it can only cope with a set of simple reductions. For example, it does not recognize reduction operations that span multiple statements, namely the "reduction group" in our compiler, as well as some other nested reductions.

Polaris [4, 13] uses a pattern-matching approach along with dependence analysis to recognize single address reductions as well as "histogram reductions"(corresponding to scalar reduction and array reductions in our model). The recognition approach is similar to ours, except that it does not recognize reductions that occur at multiple loop levels and hasn't considered reductions inside predicates.

Both SUIF and Polaris only generate code for reductions on shared memory systems, and haven't addressed the data locality and communication optimization problems.

Suganuma, Komatsu, and Nakatani [15] have implemented reduction detection and optimization in the IBM xlhpf compiler for distributed-memory machines. Their recognition algorithm identifies non-privatized variables as reduction candidates and then extracts the reduction operator and predicates for the target. They find the reduction candidates from non-privatized variables without considering array reductions.

Suganuma, Komatsu, and Nakatani generate code for distributed-memory machines, and have a very effective global communication optimization approach to coalesce and aggregate multiple communication events for reduction operations. However, they require that reduction operands be prefetchable, and then in the code generation, they transform the loop to separate the reduction operation from other computations. This excludes a set of reductions which are more closely intermixed with other computations. Forward substitution is used to merge assignments in a loop body to form a single reduction, whereas in our model, we can deal with statements at different loop levels nicely without any program transformations. Also, by forming reduction groups, we can eliminate the need for reduction coalescing or aggregation in many of the cases.

Chapter 6

Conclusions

This thesis describes reduction recognition and code generation techniques for distributed memory multiprocessors in dHPF compiler, the research compiler for data-parallel programming languages being developed at Rice University.

Study has shown that reduction operations appear in many contexts and reduction optimization is an important technique for achieving real speedups on shared memory architectures [6].

Because of the high communication cost, we expect a bigger difference between programs with or without reduction support on distributed memory machines. Our work shows that without optimization, reduction operations can become a program bottleneck; however we can get good speedups with appropriate reduction support and the time is almost negligible as compared to the non-optimized version.

With dependence analysis, pattern matching and flexible code generation support, we can handle a broad range of parallel reductions in general loop nests, even those that contain conditionals. We find that minimizing communication cost is a very important issue in generating efficient reduction code for distributed-memory machines. We use various techniques to reduce the number of collective communications, such as forming reduction groups and index encoding for MINLOC and MAXLOC reductions. To further reduce the communication, factorization is used to exploit data locality in reduction statements. We have performed experiments on IBM Scalable PowerParallel System SP2. The results indicate that without factorization, the extra communication took about 50% the time where factorization is applicable. Our compiler compares favorably to IBM's xlhpf compiler on reduction optimization, with as much as a 111% to 270% increase in efficiency on reductions for some programs for which dHPF saved a collective communication using its encoding strategy for combined MAX/MAXLOC operations.

Our techniques of exploiting data locality to reduce communication costs can also be used to generate efficient code for other recurrence parallelization techniques on distributed memory machines. For example, optimizations similar to ours can be

applied to utilize data locality in Fisher and Ghuloum's work [8, 7]. We can partition the computations of the local function compositions based on the data distributions and we may apply "factorization" to transform some composite function into the composition of a group of smaller composite functions. For example:

$$H_i = \lambda x.x + B(i-1) + C(i-1) = \lambda x.(x + B(i-1)) + C(i-1) = F_i \bullet G_i$$

where $G_i = \lambda x.x + B(i-1)$ and $F_i = \lambda x.x + C(i-1)$. Notice that the composition is commutative. If array B and C have different distributions, we can compute the operations for F_i and G_i separately to exploit data locality respectively for array B and C and then combine them to get the final results.

We have shown that reduction optimization is an important technique in parallelizing compilers. We also show that it can be implemented efficiently for distributed memory systems. However, there is still room for improvement and interesting questions to be addressed.

Future Work

Consider the multi-element reduction example in Figure 6.1. Let the data array A be of size $n \times n \times n$ and distributed with (BLOCK, BLOCK, BLOCK) distribution, and array S be of size $n \times n$ and distributed with (*, BLOCK, BLOCK) across a $p \times p \times p$ processors grid.

```
        DO i = 1, n
            DO j = 1, n
                DO k = 1, n
S1                  S(i,j) = S(i,j) + A(i,j,k)
                ENDDO
            ENDDO
        ENDDO
```

Figure 6.1 A multi-element reduction example

The code generation strategy used by the dHPF compiler as described in this thesis requires every processor to allocate a contiguous buffer space of size $n \times n$ to store the local reduction results for array S. This enables us to use a single collective

communication to get the global reduction results for all the elements in array S. However, we may have the potential problem of running out of buffer space. It requires n^2 buffer space on each of the p^3 processors. The space needed on each processor does not decrease as number of processors increases using the strategy we have discussed in this thesis.

Since the CP for statement S_1 is ON_HOME $A(i,j,k)$, the processor will not need a non-local copy of $S(i,j)$ unless it is the owner of $A(i,j,k)$ for some $1 \leq k \leq n$. Therefore, we can form subgroups of processors to compute different sections of the reduction. For example, p processors who own the array section $A(1:n/p, 1:n/p, *)$ can be grouped together to compute the reduction results for $S(1:n/p, 1:n/p)$. So each processor will only need a non-local buffer space of size n^2/p^2. It decreases quickly as p increases. However, to implement it in MPI, we need to form p^2 subgroups and invoke MPI_ALLREDUCE p^2 times, each for every single subgroup reduction. The p^2 MPI_ALLREDUCE communication calls can be performed in parallel if there is no intersection among the subgroups, while we need to maintain the correct orders to avoid deadlocks if there are intersections among the subgroups and the p_2 collective communication calls can not be executed completely in parallel. There is a conflict between resource constraints(e.g. better space availability) and communication optimization. However, it is necessary to consider machine-dependent resource constraints to ensure the correctness of the communication placement. Some of the resource-based communication placement techniques developed by Kennedy and Sethi [12] can be adapted here to further analyze and solve the problem. However, when there are strided or irregular array accesses, the difficulties of forming subgroups and handling possibly a large number of collective communications may further complicate the problem.

Appendix

There are 6 figures in this appendix.

Figure A-1 through A-3 list the code generated for erlebacher reduction kernel by dHPF without reduction support. There are `send`s and `recv`s between every two processors(lines 11 to 72) to provide each processor with a full local copy of array `duz`. After receiving all the data, every processor will do the computation and get the final results(lines 76 to 130).

Figure A-4 through A-6 list the code generated for tomcatv reduction kernel by xlhpf compiler. From lines 9 to 17, it iterates over the local elements and get the local maximum absolute value. At line 22, it invokes "_xlhpf_reduce_maxval" to get the global maximum absolute value and store it in `rxm`. Then, from lines 28 to 39, it builds another loop to get the location for the local maximum absolute value. From lines 44 to 50, it invokes "_xlhpf_reduce_maxloc" to get the location coordinates `irxm` and `jrxm` for the global maximum absolute value. In the second half of the program, it goes through the same process to get the results for `rym, irym`, and `jrym`.

```
        ...

        do j = 1, 64
          do i = 1, 64
            tot(i, j) = 0.
          enddo
        enddo
C
C       Loop section ---[ 0 <= p_q1 <= 3 ]---
C
11      do p_q1 = 0, 3
          if (p_myid1 .ne. p_q1) then
            p_coord_comm(1) = p_q1
            call mpi_cart_rank(p_cmap, p_coord_comm(1), q_id_comm, ierr)
            call mpi_cart_coords(p_cmap, q_id_comm, 1, p_coord_comm(1),
     *ierr)
            p_q1 = p_coord_comm(1)
C           ***** MPI SEND STMT FOR NONLOCAL READ *****
            counter_duz_2 = 0
            lb = max(p_myid1 * 16 + 1, 1) * 4096 - p_myid1 * 65536 - 409
     *6
            ub = min(p_myid1 * 16 + 16, 63) * 4096 - p_myid1 * 65536 - 1
            counter_duz_2 = ub - lb + 1
            if (counter_duz_2 .gt. 0) then
              call mpi_bsend(duz(1, 1, max(p_myid1 * 16 + 1, 1) - p_myid
     *1 * 16 - 1), counter_duz_2, mpi_real, p_q1, 1, p_cmap, ierr)
            endif
          endif
        enddo
C
C       Loop section ---[ 0 <= p_q1 <= 3 ]---
C
        do p_q1 = 0, 3
          if (p_myid1 .ne. p_q1) then
            p_coord_comm(1) = p_q1
            call mpi_cart_rank(p_cmap, p_coord_comm(1), q_id_comm, ierr)
            call mpi_cart_coords(p_cmap, q_id_comm, 1, p_coord_comm(1),
     *ierr)
            p_q1 = p_coord_comm(1)
C           --< Loop Counters >--
            counter_duz_2 = 0
            min_result_0__dhpf = min(16 * p_q1 + 16, 63)
```

Figure A-1 Code generated for erlebacher reduction kernel without reduction support, Part 1 of 3

```
            if (16 * p_q1 + 1 .le. min_result_0__dhpf) then
              counter_duz_2 = counter_duz_2 + 64 * 64 * (min(16 * p_q1 +
     * 16, 63) - (16 * p_q1 + 1) + 1)
            endif
            call hpf_buffer_alloc(counter_duz_2 * 4, recv_buf_duz_2)
            call hpf_ptr_to_index(hpf_heap, recv_buf_duz_2, 4, recv_buf_
     *duz_2_index)
            if (counter_duz_2 .gt. 0) then
              call mpi_recv(hpf_heap_real(recv_buf_duz_2_index), counter
     *_duz_2, mpi_real, p_q1, 1, p_cmap, status, ierr)
            endif
C           --< Unpack Loop From Recv For Nonlocal Read >--
            counter_duz_2 = 0
C
C           Loop section ---[ ((16 * p_q1) + 1) <= i3 <= min(((16 * p_q1) + 16),
C 63), 1 <= i2 <= 64, 1 <= i1 <= 64 ]---
C
            do i1 = 1, 64
              do i2 = 1, 64
                i3_end__dhpf = min(16 * p_q1 + 16, 63)
                do i3 = 16 * p_q1 + 1, i3_end__dhpf
                  call hpf_nonlocal_insertr(hash_nonlocals, duz_data, i3
     * * 4096 + i2 * 64 + i1 - 4161, hpf_heap_real(recv_buf_duz_2_index
     *+ counter_duz_2))
                  counter_duz_2 = counter_duz_2 + 1
                enddo
              enddo
            enddo
            call hpf_buffer_free(recv_buf_duz_2)
          endif
72       enddo
C
C        Loop section ---[ 1 <= k <= 63 ]---
C
76       do k = 1, 63
C
C          --<< Iterations that access only local values >>--
C
C
C          Loop section ---[ 1 <= i <= 64, 1 <= j <= 64 ]---
C
```

Figure A-2 Code generated for erlebacher reduction kernel without reduction support, Part 2 of 3

```
          if (16 * p_myid1 .le. k - 1 .and. 16 * p_myid1 .ge. k - 16) th
     *en
            do j = 1, 64
              do i = 1, 64
                tot(i, j) = tot(i, j) + d(k) * duz(i, j, k - p_myid1 * 1
     *6 - 1)
              enddo
            enddo
          endif

C
C         --<< Iterations that read (but do not compute) non-local values >>--
C
C
C         Loop section ---[ 1 <= i <= 64, 1 <= j <= 64 ]---
C
          if (16 * p_myid1 .ge. k) then
            do j = 1, 64
              do i = 1, 64
                if (p_myid1 .eq. (k - 1) / 16) then
                  lnltmp1 = duz(i, j, k - p_myid1 * 16 - 1)
                else
                  lnltmp1 = hpf_nonlocal_lookupr(hash_nonlocals, duz_dat
     *a, k * 4096 + j * 64 + i - 4161)
                endif
                tot(i, j) = tot(i, j) + d(k) * lnltmp1
              enddo
            enddo
          endif
C
C         Loop section ---[ 1 <= i <= 64, 1 <= j <= 64 ]---
C
          if (16 * p_myid1 .le. k - 17) then
            do j = 1, 64
              do i = 1, 64
                if (p_myid1 .eq. (k - 1) / 16) then
                  lnltmp0 = duz(i, j, k - p_myid1 * 16 - 1)
                else
                  lnltmp0 = hpf_nonlocal_lookupr(hash_nonlocals, duz_dat
     *a, k * 4096 + j * 64 + i - 4161)
                endif
                tot(i, j) = tot(i, j) + d(k) * lnltmp0
              enddo
            enddo
          endif
        enddo
130 ...
```

Figure A-3 Code generated for erlebacher reduction kernel without reduction support, Part 3 of 3

```
      ...
       w = 1d0 + 0d0
       IVAINIT_72 = w
C 1585-501  Original Source Line 34
       do I_39=1d0,(100d0 - 1d0 + 1d0) / 1d0,dble(1)
         rxm = -1.7976931348623157d+308
         SCALAR_43 = -1.7976931348623157d+308
C 1585-501  Original Source Line 35
9        do i_12=1,514,1
C 1585-501  Original Source Line 35
           do i_13=iown_l_24,MIN0(iown_u_25,514),1
C 1585-501  Original Source Line 35
             if ((DABS(rx_61(i_13,i_12)) .gt. SCALAR_43) .ne. 0) then
               SCALAR_43 = DABS(rx_61(i_13,i_12))
             end if
           end do
17       end do
         Recv_index_44(1) = (-2)
         DS_SAS_45(1) = 0
         DS_SAS_45(2) = 3
         DS_SAS_45(3) = 1
22       call _xlhpf_reduce_maxval(10,SCALAR_43,rxm,PG_23,1,0,DS_SAS_45,
     &Recv_index_44)
         T_17 = -1.7976931348623157d+308
         T_18_66(1) = 0
         T_18_66(2) = 0
C 1585-501  Original Source Line 36
28       do i_12=1,i2m - i1p + 1,1
C 1585-501  Original Source Line 36
           do i_13=iown_l_24,MIN0(iown_u_25,m),1
C 1585-501  Original Source Line 36
             if ((DABS(rx_61(i_13,i_12 + (i1p - 1))) .gt. T_17) .ne. 0)
     &then
               T_17 = DABS(rx_61(i_13,i_12 + (i1p - 1)))
               LOC_46(1) = i_13
               LOC_46(2) = i_12
             end if
           end do
39       end do
         Recv_index_49(1) = (-2)
         DS_SAS_50(1) = 0
         DS_SAS_50(2) = MIN0((m - 1) / 129,3)
         DS_SAS_50(3) = 1
44       call _xlhpf_reduce_maxloc(10,2,LOC_46,T_17,%val(T_18),PG_23,1,0
     &,DS_SAS_50,Recv_index_49)
```

Figure A-4 Code generated for tomcatv reduction kernel by xlhpf compiler, Part 1 of 3

```
C 1585-501  Original Source Line 36
         do i_12=1,2,1
           max_locx_64(i_12) = T_18_66(i_12)
         end do
         irxm = max_locx_64(1)
50       jrxm = max_locx_64(2)
         rym = -1.7976931348623157d+308
         SCALAR_51 = -1.7976931348623157d+308
C 1585-501  Original Source Line 40
         do i_12=1,514,1
C 1585-501  Original Source Line 40
           do i_13=iown_l_24,MIN0(iown_u_25,514),1
C 1585-501  Original Source Line 40
             if ((DABS(ry_62(i_13,i_12)) .gt. SCALAR_51) .ne. 0) then
               SCALAR_51 = DABS(ry_62(i_13,i_12))
             end if
           end do
         end do
         Recv_index_52(1) = (-2)
         DS_SAS_53(1) = 0
         DS_SAS_53(2) = 3
         DS_SAS_53(3) = 1
         call _xlhpf_reduce_maxval(10,SCALAR_51,rym,PG_23,1,0,DS_SAS_53,
     &Recv_index_52)
         T_19 = -1.7976931348623157d+308
         T_20_67(1) = 0
         T_20_67(2) = 0
C 1585-501  Original Source Line 41
         do i_12=1,i2m - i1p + 1,1
C 1585-501  Original Source Line 41
           do i_13=iown_l_24,MIN0(iown_u_25,m),1
C 1585-501  Original Source Line 41
             if ((DABS(ry_62(i_13,i_12 + (i1p - 1))) .gt. T_19) .ne. 0)
     &then
               T_19 = DABS(ry_62(i_13,i_12 + (i1p - 1)))
               LOC_54(1) = i_13
               LOC_54(2) = i_12
             end if
           end do
         end do
         Recv_index_55(1) = (-2)
         DS_SAS_56(1) = 0
         DS_SAS_56(2) = MIN0((m - 1) / 129,3)
         DS_SAS_56(3) = 1
         call _xlhpf_reduce_maxloc(10,2,LOC_54,T_19,%val(T_20),PG_23,1,0
     &,DS_SAS_56,Recv_index_55)
```

Figure A-5 Code generated for tomcatv reduction kernel by xlhpf compiler, Part 2 of 3

```
C 1585-501  Original Source Line 41
          do i_12=1,2,1
            max_locy_65(i_12) = T_20_67(i_12)
          end do
          irym = max_locy_65(1)
          jrym = max_locy_65(2)
          w = w + 1d0
        end do
        ...
```

Figure A-6 Code generated for tomcatv reduction kernel by xlhpf compiler, Part 3 of 3

Bibliography

[1] V. Adve, J. Mellor-Crummey, and A. Sethi. HPF analysis and code generation using integer sets. Technical Report CS-TR97-275, Dept. of Computer Science, Rice University, April 1997.

[2] 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.

[3] R. Barua, D. Kranz, and A. Agarwal. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.

[4] W. Blume et al. Effective automatic parallelization with polaris. In *International Journal of Parallel Programming*, May 1995.

[5] E. Darnell. Special reductions in PFC. Supercomputer Software Newsletter 13, Dept. of Computer Science, Rice University, October 1986.

[6] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[7] A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[8] A. Ghuloum and A. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[9] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[10] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Interprocedural parallelization analysis: Preliminary results. Technical Report CSL-TR-95-665, Dept. of Computer Science, Stanford University, March 1995.

[11] K. Kennedy and R. Allen. *Advanced Compilation for Vector and Parallel Computers*. Morgan Kaufmann Publishers, San Mateo, CA, 1997.

[12] K. Kennedy and A. Sethi. Resource-based communication placement analysis. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.

[13] B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. CSRD Rpt. No. 1396, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January 1995.

[14] 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.

[15] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.