Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors*

Bo Lu and John Mellor-Crummey
Department of Computer Science
Rice University
{bolu, johnmc}@cs.rice.edu

Abstract

This paper presents reduction recognition and parallel code generation strategies for distributed-memory multiprocessors. We describe techniques to recognize a broad range of implicit reduction operations, including those involving statements at multiple loop nesting levels and intermixed with conditional control flow. We introduce two new optimizations: factoring *which increases data locality for* SUM *and* PRODUCT *reductions, and* index encoding *which enables a single global communication to accomplish both an extreme value reduction and an extreme value location reduction. We have implemented these techniques in the dHPF compiler for High Performance Fortran (HPF). We evaluate their effectiveness experimentally by compiling several reduction benchmarks with dHPF and two commercial HPF compilers, and comparing the performance of the generated code on an IBM SP2. Our results show that our recognition techniques are more powerful and that our* index encoding *and* factoring *optimizations can improve performance by a factor of two where they apply.*

1. Introduction

High Performance Fortran (HPF) [10, 12] provides an attractive model for parallel programming because of its relative simplicity. Principally, programmers write a single-threaded Fortran program and use data layout directives to map data elements onto an array of processors. HPF compilers use these directives to partition a program's computation among processors, and to synthesize necessary data movement and synchronization. One can parallelize an application written in sequential Fortran simply by adding HPF directives to the program and leaving it to an optimizing compiler to analyze, understand and exploit parallelism inherent in the program. However, to achieve high performance for scientific programs with this approach, a compiler must recognize and generate efficient code for reductions.

A reduction is a commutative and associative operation that maps an array of n dimensions to an array of m dimensions, where $0 \leq m < n$[1]. Reduction operations appear in many contexts including kernels for matrix multiplication, image processing, computational geometry algorithms, sorting, and are commonly used to test for convergence of iterative algorithms.

Performing reductions on distributed-memory multiprocessors is costly because of the need for communication that may involve all of the processors. When compiling programs in which reductions are coded implicitly, an optimizing compiler without explicit support for reduction recognition will naively classify reductions as sequential operations. This happens because data dependence analysis will identify the reduction's repeated updates of the same location as an obstacle to parallel execution. However, reduction computations can be parallelized since commutative updates to an accumulator can be reordered safely.

Reduction computations can be optimized effectively for a variety of architectures. On MIMD parallel computers, a reduction can be computed efficiently by having each processor (in parallel) compute a partial reduction result and then combining these partial results into a final result by using one or more collective communication operations. In this manner, a parallel reduction of n data elements into m data elements on p processors can be computed in time $O(n/p + m \log p)$ (assuming that the data is evenly distributed). To generate reduction code that achieves high performance on distributed-memory machines, it is important minimize communication which can be costly on such systems.

Many of the previous approaches for recognizing reductions by detecting recurrences are principally based on

*Appeared in *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

[1] An array of zero dimensions represents a scalar value.

pattern matching. The SUIF [9] compiler uses a pattern-matching strategy to recognize commonly occurring reductions. While it recognizes only simple reductions, it can recognize them interprocedurally. Polaris [4, 13] can recognize single address reductions, which accumulate their result into a scalar variable, and histogram reductions, which accumulate their results into an array. Both SUIF and Polaris generate code for parallel reductions on shared memory systems, and don't address the issues of data locality and communication optimization which are key for distributed-memory systems.

Fisher and Ghuloum [7, 8] describe a technique for recognizing reductions and scan operations that is much more powerful than those supported by other parallelizing compilers. However, their recognizer implicitly assumes that loops with recurrences are not intermixed with other code. They generate code for iWarp, a distributed-memory MIMD supercomputer. Their code generation model is simple and does not optimize data locality and communication. Their techniques are especially useful for parallelizing complex scan operations, but reductions, as a special case, are not handled as efficiently as in our compiler.

The IBM HPF compiler [16] recognizes only reductions into scalar variables. It generates code for distributed-memory machines and optimizes reductions by coalescing and aggregating communication together for multiple reductions. A limitation of this work is that it relies on forward substitution and reduction operand prefetching which limits reduction optimization when a reduction is intermixed with other computation.

Reduction recognition in the dHPF compiler grew out of earlier work in PFC [6], which recognized implicit SUM and PRODUCT reductions to convert them into equivalent explicit FORTRAN 90 intrinsic calls. In dHPF, we extended this work to handle more general forms of SUM and PRODUCT reductions, developed support for recognition of MIN, MAX, MINLOC, and MAXLOC reductions, and developed code generation support for distributed memory machines. Contributions of this work include:

- *A reduction recognition technique that recognizes a variety of multi-level reductions intermixed with other computation.*
- *An efficient code generation strategy that uses techniques including* reduction grouping *and* index encoding *to reduce communication on distributed-memory machines.*
- *A* factoring *transformation that exploits the associativity and commutativity of reduction operations to reduce communication.*

As our experiments show in section 4, our reduction recognition strategy is effective and our optimization strategies can improve performance for reductions by a factor of two where they apply.

The rest of this paper is organized as follows. In section 2, we describe our reduction recognition algorithms. In section 3, we describe reduction optimization and code generation for distributed memory machines. In section 4, we evaluate the effectiveness of these techniques by comparing the performance of code generated by dHPF and with the performance of code generated by HPF compilers from IBM and the Portland Group. We summarize our conclusions in section 5.

2. Reduction Recognition

In this section, we describe how to recognize SUM, PRODUCT, MIN, MAX, MINLOC and MAXLOC reduction patterns. There are two phases to reduction recognition. First, for each assignment statement, we check if it is reducible and if so, classify its reduction type. Second, we gather related reduction statements into 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.

2.1. Single Statement Reductions

To identify whether an assignment statement is part of a reduction computation, we use four program representations: an abstract syntax tree (AST), a data dependence graph [11], static single assignment (SSA) form [5] and a control dependence graph [5]. We first identify assignment statements with incident *true*, *anti* and *output* data dependences as reduction candidates. Next, we use SSA to locate definitions of the right hand side variables and use the AST to inspect the operations on right hand side variables in reduction candidates. For extreme value reductions formed by MIN, MAX, MINLOC, or MAXLOC operations, we use control dependence information to find a control dependence predecessor and compare the control statement with the dependent statements to determine if they form an extreme value reduction. The use of control dependence information distinguishes our method from others, and enables us to recognize extreme value reductions using various control flow structures.

2.2. Reduction Groups

For SUM or PRODUCT reductions, a reduction group is formed by reduction statements with the same left hand side accumulator and same reduction operator. For example, in the following code fragment

```
do i = lb, ub
  T = T + Y(i)
  T = T + Z(i, n)
enddo
```

both of the assignments to T belong to the same SUM reduction group with the accumulator of T. For MIN/MAX reductions, a reduction group is formed by the assignment statement that records the extreme value along with any assignment statements that record the position of the extreme value.

All statements in a reduction group use the same storage for accumulating the local reduction results (a local accumulator for T in the example above), and use a single collective communication operation to compute the global results. Our approach differs from the reduction handling in the IBM HPF compiler [16]. They first generate a separate communication event for each reduction statement, and then apply reduction coalescing and aggregation to merge communication operations where possible. By forming a reduction group and generating one communication event for each group, we avoid the need for communication coalescing in most cases.

For any pair of statements to be members of the same reduction group, their common accumulator must not be modified or referenced by non-reduction statements at their deepest common loop level. In the following example,

```
    do i = lb, ub
S1      T = T + Y(i)
S2      T = T + Z(i,n)
S3      Q(i) = T
    enddo
```

statement S_3 is not a reduction candidate and it references T at the same loop level as S_1 and S_2, thus we can't form a reduction group with S_1 and S_2. However, if T is only referenced or modified by non-reduction statements outside the loop level at which the reduction candidates' dependences are carried, the candidates belong to the same reduction group.

Reduction Levels

The dHPF compiler recognizes multi-level reduction groups. Statements in a reduction group need not all be at the same loop nesting level. We use the data dependence graph to detect modifications and references to a reduction accumulator by non-reduction statements to decide how many loop levels can participate in a reduction. For example, the following loop nest

```
    do k = 1, ub1
S1    Q(k) = S+9
S2    S = S+B(k)
      do i = 1, ub2
G1      S = S+C(i)
        do j = 1, upb3
G1        S = S+E(j)
        enddo
G1      S = S+D(i)
      enddo
    enddo
```

contains one reduction group G_1 that contains the statements as labeled. Since S_1, a non-reduction statement, references the value of S at the same loop level as reduction candidate S_2, S_2 cannot be a member of any reduction group. However, since the intermediate values of S are not referenced by a non-reduction candidate inside the i or j loops, all of the statements marked G_1 are collected into the same reduction group. The reduction for these inner two loops can be computed in parallel to provide the value of S used by S_1. We say group G_1 is reducible in the inner two loops. Figure 1 describes the process for building reduction groups.

Input: a Fortran 77 program and basic program analysis.

Output: the reduction groups in the program.

1. Build reduction groups for statements directly inside a loop.
   (a) Identify assignment statements that meet the criteria for being a reduction candidate (as described in section 2.1). Classify the type of the reduction by examining the contributing operators and the control dependences.
   (b) For each set of reduction candidates that have the same accumulator and reduction type, collect them into a reduction group as long as any other uses or modifications of the accumulator occur outside the innermost loop level at which the reduction dependences are carried.
2. Merge reduction groups at different loop levels. If there are two reduction groups, G_1 at level m and G_2 at level n, where $m < n$, merge G_1 and G_2 if they have the same accumulator and reduction type and no non-reduction candidate accesses the accumulator at levels m through n, inclusive. Otherwise, if the reduction accumulator is accessed at level k, where $m \leq k < n$, group G_1 in the outer loop is not reducible and we cannot merge the two groups.

   To compute whether groups can be merged, for each reduction group G_1 at level m:
   (a) Search all of the dependences of the accumulation variable of G_1 to find other statements that reference or modify 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 outermost level at which the reduction can be performed based on the dependences checked in step 2a.

Figure 1. Algorithm for building reduction groups.

2.3. Idioms Recognized

If implicit reductions are not recognized and parallelized, they can dramatically degrade parallel program performance. With suitable optimization, the impact on performance can be reduced substantially. For this reason, it is very important to have powerful support for recognizing reductions. In dHPF, the algorithms described earlier in this section can recognize a broad range of reduction operations. Here we provide a brief enumeration of the reduction features that dHPF can recognize.

- Scalar reductions which accumulate the results into a scalar variable.
- Multi-element array reductions which accumulate reduction results into one or more elements in an array.
- Reduction groups that may contain multiple reduction statements (each of which may be at a different loop nesting level) sharing the same accumulator.
- Reduction operations that are control dependent on conditionals such as

  `if` α `then s = s * a(i)`

  can form a reduction group with other statements outside the control statement.
- MIN/MAX and MINLOC/MAXLOC reductions using different forms of `if` structures. Absolute value operations are fully supported for MIN/MAX or MINLOC/MAXLOC operations as we demonstrate in section 4 with an example from the SPEC92 Tomcatv benchmark program.
- Reductions closely intermixed with other computations. For example, dHPF can recognize reductions that use arrays or privatizable variables which are defined previously in the same loop; other compilers require that reduction operations be isolated from other computations and that their operands be *prefetchable* [16].

3. Reduction Code Generation

Compiling data parallel programs to distributed-memory machines consists of two major phases. The first phase determines how to decompose the computation and data across the processors in a fashion that not only parallelizes the application, but also minimizes communication. The dHPF compiler uses the data layout directives in an HPF program to select a separate computation partitioning for each program statement. A computation partitioning specifies which processor or processors will execute each dynamic instance of that statement. The second phase of compilation uses the computation partitioning to generate SPMD code so that each processor executes its allotted computation and communication correctly and efficiently. Here we describe the version of dHPF that generates message passing code for an MPI [15] communication substrate, although the techniques apply to distributed shared memory systems as well.

3.1. Computation Partitioning

The dHPF compiler 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.

There are three steps in parallelizing a reduction operation. Here, we use a SUM reduction consisting of a loop containing the statement `S = S + A(i)` as an example. In a reduction preamble, each processor stores the original value of `S` into a temporary variable `T`, and initializes `S` to be zero. In the reduction core, each processor owning a part of array `A` involved in the reduction computes the partial sum its local values into `S`. Finally, in a postamble, the processors accumulate their partial sums using a collective communication operation, and add back the 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 producing the final reduction value using collective communication. 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. Factorization and Data Locality

Suppose we have a reduction statement $S = S \oplus A_1(f_1(\vec{i})) \oplus A_2(f_2(\vec{i})) \ldots \oplus A_n(f_n(\vec{i}))$ in a loop nest where $\vec{i}$ is the vector of enclosing loop indices, $n > 1$, and $\oplus$ is a commutative and associative operator. If some of the arrays $A_1 \ldots A_n$ are distributed differently, some instances of this statement will need to read off-processor data no matter how we specify their CP. For example, suppose we compute the above statement ON_HOME $A_1(f_1(\vec{i}))$. If $A_2(f_2(\vec{i}))$ is not always local to all of the processors owning values referenced by $A_1(f_1(\vec{i}))$, some processors owning elements of $A_1(f_1(\vec{i}))$ will need to read non-local values for $A_2(f_2(\vec{i}))$ to compute the partial sum for their assigned statement instances.

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

$$\begin{array}{ll} S = S \oplus A_1(f_1(\vec{i})), & S = S \oplus A_2(f_2(\vec{i})), \\ \ldots, & S = S \oplus A_n(f_n(\vec{i})) \end{array}$$

After this transformation, we can compute each reduction statement $S = S \oplus A_k(f_k(\vec{i}))$ $(1 \leq k \leq n)$ ON_HOME $A_k(f_k(\vec{i}))$ without communication. Our compilation model accommodates the factorization process in a natural fashion. Each of the simple statements factored out of a complex statement will be in the same reduction group whose final result will be accumulated by a single collective communication operation in the postamble.

3.3. Sum/Product Reductions

Handling of scalar reductions is straightforward. Since dHPF replicates storage for scalars across each processor, we can use the local instance of a scalar to store the local reduction value. In the preamble, we save the original value of the scalar into a scalar temporary variable and initialize the local reduction value to the identity element Φ[2]. In the reduction core, we compute a partial reduction result on each processor using only its local data. In the postamble, we call MPI_ALLREDUCE to combine all of the partial reduction results from different processors. Finally, each processor combines the result from MPI_ALLREDUCE with the reduction variable's original value (saved in the preamble) to get the final result.

For a multi-element array reduction, consider an array distributed among the processors. On each processor, we allocate a contiguous buffer large enough to store the elements of the local reduction result. In the preamble, we initialize each element in this buffer to Φ. In the reduction core, we compute the partial reduction results into the buffer using contributions from local data. In the postamble, a single call to MPI_ALLREDUCE combines the partial results across the processors for all of the buffer elements in the reduction. Finally, for every element in the result array, the owner(s) of that element is (are) responsible for combining the original value with the result of MPI_ALLREDUCE to produce the final result.

3.4. Extreme Value Reductions

Code generation for a MIN/MAX reduction is similar to that for SUM/PRODUCT scalar reductions, except that the initial value participates in the local MIN/MAX selections, and the global result from MPI_ALLREDUCE is the final result.

The semantics of MPI_ALLREDUCE for reduction types MPI_MINLOC or MPI_MAXLOC is to compute a global minimum or 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 one coordinate index in a 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 an ordered pair (local extreme value, coordinate index of local extreme value), and apply the appropriate MPI_MINLOC/MPI_MAXLOC reduction operation to get the global extreme value result along with the minimum index at which the extreme value was found.

In cases such as the following loop nest in which multiple coordinate indices are needed,

```
      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
```

we have each processor *encode* the coordinates of its local extreme value into a linearized coordinate space. For the above example, we use the encoding

[2] Φ is 0 for SUM reduction and 1 for PRODUCT reduction.

$$((\text{maxk-lb}_k)\times(\text{ub}_j\text{-lb}_j)+(\text{maxj-lb}_j))\times(\text{ub}_i\text{-lb}_i)+(\text{maxi-lb}_i)$$

where lb_k represents the lower bound in k's iteration space, ub_k is the upper bound, and similarly for $\text{lb}_j, \text{ub}_j, \text{lb}_i$, and ub_i. Combining the local extreme values and the encoded coordinates with an MPI_MAXLOC operation computes the global MAX result and the minimum encoded coordinate value at which the extreme value was found. We decode the coordinates for the final result. This encoding strategy computes both the extreme value and its location in a single collective communication.

Using the above encoding strategy, if two or more elements have the same maximum value, the encoded coordinate result provides the smallest coordinates (k,j,i) in dictionary order. A more complex encoding strategy is needed to accommodate extreme value reduction loops for the full range of extreme value comparison operators (`LT`,`LE`,`GT`,`GE`) with a mixture of loops in ascending and descending 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" determines 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. For this example, we encode the coordinates as

$$(\text{ub}_j\text{-maxj})\times(\text{ub}_i - \text{lb}_i) + (\text{maxi-lb}_i).$$

With this encoding, after a global MPI_MAXLOC operation, we obtain the largest `j` and the smallest `i`, just as we would with a sequential execution order.

In general, for a loop nest n loops, $1 \leq i \leq n$, where $i = 1$ is the innermost loop, the encoding is

$$c_1 + \sum_{i=2}^{n} c_i \prod_{j=1}^{i-1} (\text{ub}_j - \text{lb}_j) \qquad (1)$$

where ub_i and lb_i are the upper bound and lower bound values for the induction variable at loop height i, and as shown in figure 2, the coefficient c_i for the the loop at height i is determined by the comparison operator and whether the loop traverses iterations in ascending or descending order.

To accommodate extreme value reductions that test the absolute value of the extreme value but record the signed extreme value (see figure 3), we use the magnitude of the extreme value for the global MPI_MINLOC or MPI_MAXLOC collective communication and augment the low bit of the coordinate encoding to carry the sign of the extreme value. For example, for each of the reductions in the tomcatv kernel shown in figure 3, we apply the MPI_MAXLOC operation to the pair (local absolute max, 2×*encodedCoordinates*+*sign*) on each processor, where "local absolute max" is the maximum absolute value on each processor, *encodedCoordinates* corresponds to the encoding of equation 1, and *sign* is a bit that represents the sign of the element that contributes the maximum absolute value on each processor. We arbitrarily use "0" for positive a number and "1" for negative number. Following the global collective communication, we decode the results to recover the absolute maximum, its coordinates, and its sign. Since we augmented the coordinate encoding with the sign in the lowest bit, it will not change the coordinate encoding selected.

Using the encoding strategy, we can invoke a single MPI_ALLREDUCE call to get both the extreme value and its coordinates.

Inputs: `compOp`: the comparison operator for loop *i*,
`ascending`: whether the loop is in ascending order
`coordVar`: var containing the extreme value coordinate
`lb`: the lower bound of the loop
`ub`: the upper bound of the loop
Output: `coefficient`: coordinate encoding coefficient

```
Boolean first := (compOp∉{GE,LE} ?true,false);
Boolean smallest :=
  (ascending && first) || (!ascending && !first);
if(smallest)
  coefficient := create_minus_expr(coordVar, lb);
else
  coefficient := create_minus_expr(ub, coordVar);
```

Figure 2. Pseudo-code to generate the extreme value coefficient term for a loop level.

4. Evaluation

We compare the performance of code generated by dHPF against that of two commercial HPF compilers, IBM *xlhpf* and PGI *pghpf*. IBM *xlhpf* 1.02 is a High Performance Fortran compiler for machines running the AIX operating system such as the IBM SP2. PGI *pghpf* 2.2 is the Portland Group's implementation of HPF that generates code for a variety of platforms including the IBM SP2. Our performance comparison is based on the performance of compiler-generated code for three implicit reduction benchmarks: an extreme value reduction benchmark using a kernel from the SPEC92 version of Tomcatv, a multi-dimensional array reduction from the Erlebacher benchmark, and a synthetic benchmark to show the benefits of our factorization optimization. We discuss each of these examples in turn in the following subsections.

The experimental platform for our comparisons is a 64-processor IBM Scalable PowerParallel System SP2. The dHPF compiler-generated code communicates using the IBM implementation of the message passing interface (MPI) communication standard on top of the SP2 user space (us) communication subsystem. The code generated by IBM's *xlhpf* uses a run-time support library based on IBM's MPL message passing library that is the communication layer underlying their MPI implementation. The code generated by PGI's *pghpf* uses PGI's transport-independent interface.

4.1. Tomcatv Reduction Kernel

Tomcatv is a 195-line mesh generation program from the SPEC92 floating-point benchmark suite. It contains two groups of MAXVAL, MAXLOC reduction operations. The code in Figure 3 shows the reduction kernels extracted from the full benchmark. Our Tomcatv reduction benchmark, T77, contains a timestep loop that executes this kernel 100 times. The `rx` and `ry` arrays are distributed across the processors in a 1-D `(BLOCK, *)` distribution. In each iteration of the timestep loop, our benchmark also performs an embarrassingly parallel assignment to arrays `rx` and `ry` in addition to the reductions so that the reduction results do not appear loop invariant (to ensure that the compilers do not hoist the reductions outside the timestep loop).

The dHPF compiler recognizes the two groups of extreme value reduction operations in the T77 kernel. For each reduction group, dHPF generates code so that each processor computes the extreme value and coordinate location for its local data, and then uses the coordinate encoding strategy described in section 3.4 with a single collective call to MPI_ALLREDUCE to perform an MPI_MAXLOC operation that computes the global absolute maximum value, value sign, and coordinate location.

Neither the *xlhpf* compiler nor the *pghpf* compiler recognize the implicit reductions in the T77 kernel. Given the T77 kernel, they generate code that propagates arrays `rx` and `ry` to each processor and computes the full reductions sequentially on each processor. Because of the high cost of disseminating all of the values in `rx` and `ry` to each processor, this generated code scales poorly. Table 1 shows the timings for code generated for the T77 benchmark kernel by each of the compilers for a range of processor numbers.

To get a more reasonable performance baseline for comparing the performance of dHPF's code for the T77 benchmark kernel, we recoded the T77 benchmark kernel to use the MAXLOC FORTRAN 90 reduction intrinsic that both

```
    DO 270 j = 1,m
    DO 270 i = i1p,i2m
    IF (abs(rx(i,j)) .LT. ABS(rxm))  GOTO  262
    rxm  = rx(i,j)
    irxm = i
    jrxm = j
262 IF (ABS(ry(i,j)) .LT. ABS(rym))  GOTO  270
    rym  = ry(i,j)
    irym = i
    jrym = j
270 CONTINUE
```

Figure 3. T77: FORTRAN 77 Tomcatv Reduction Benchmark Kernel.

```
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)
```

Figure 4. T90: FORTRAN 90 Tomcatv Reduction Benchmark Kernel.

xlhpf and *pghpf* handle as a data-parallel reduction. The refined T90 kernel is shown in figure 4. In each reduction group, we use the `maxloc` intrinsic to get the coordinates of the absolute extreme value, then use the coordinates to recover the signed extreme value. For the T90 benchmark, both *xlhpf* and *pghpf* generate two collective communication operations for each reduction. For each of `rx` and `ry`, they generate a collective communication to compute the global MAXLOC, along with a broadcast to propagate the max value to all of the processors. The T90 rows in table 1 show the times for *xlhpf* and *pghpf* on the explicit T90 reduction benchmark. Figure 5 shows the best speedup (the T77 or T90 benchmark as appropriate) for each of the three compilers.

Comparing both the tabular and graphical results for the Tomcatv reduction kernel performance results shows that the dHPF compiler achieves the highest performance with the T77 benchmark and generates code that is more than a factor of two faster than either the *pghpf* or *xlhpf* compilers, even when the benchmark is recoded in FORTRAN 90 for their benefit. When the *pghpf* and *xlhpf* compilers do not recognize the reductions in the T77 benchmark, the performance degradation is catastrophic, costing from a factor of

20 to several thousand.

nprocs		2	4	8	16	32
dHPF	T77	2.66	1.59	0.84	0.47	0.34
xlhpf	T77	48.0	89.5	90.9	106.6	116.7
pghpf	T77	664	2852	2651	3249	4274
xlhpf	T90	5.07	3.72	3.07	2.80	2.81
pghpf	T90	4.93	2.63	1.49	0.93	0.72

Table 1. Execution time(sec) for the T77 and T90 Tomcatv reduction kernels.

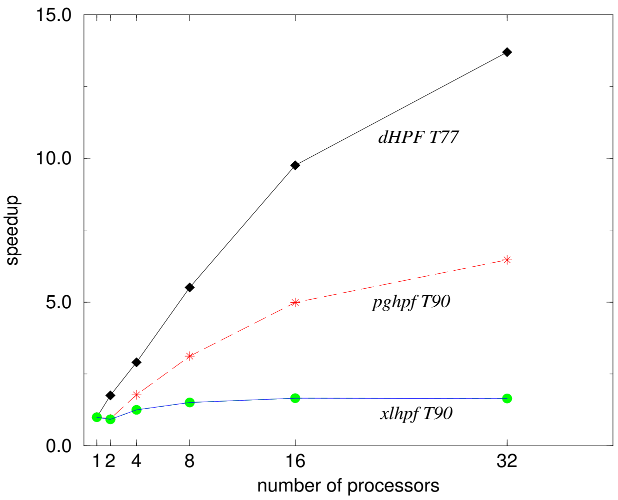

Figure 5. Speedups for T77 and T90 reduction kernels.

4.2. Erlebacher Reduction Kernel

Erlebacher is an 600 line, ten procedure benchmark program from ICASE that performs 3D compact differencing. It includes a number of fully parallel phases interleaved with multi-dimensional reductions and computational wavefronts that perform forward and backward substitutions. We tested the compilers on the multi-dimensional reduction kernel extracted from Erlebacher as shown in figure 6.

Since *xlhpf* doesn't have support for array reductions, it executes the above operation sequentially on every processor. We compared dHPF with *pghpf* compiler. They both generate parallel reduction code for the kernel. Unlike dHPF, which generates code to invoke a single collective communication reduction to combine the local sums on each processor for the entire array `tot`, *pghpf* invokes a

```
      PARAMETER (n=64)
      DO 40 k=1,n-1
        DO 40 j=1,n
          DO 40 i=1,n
            tot(i,j) = tot(i,j) + d(k)*duz(i,j,k)
 40       CONTINUE
```

Figure 6. Erlebacher reduction kernel.

global reduction call for each element in the array, with a total of 64×64 reduction calls. Figure 7 shows the speedups for Erlebacher reduction kernel using dHPF and *pghpf*.

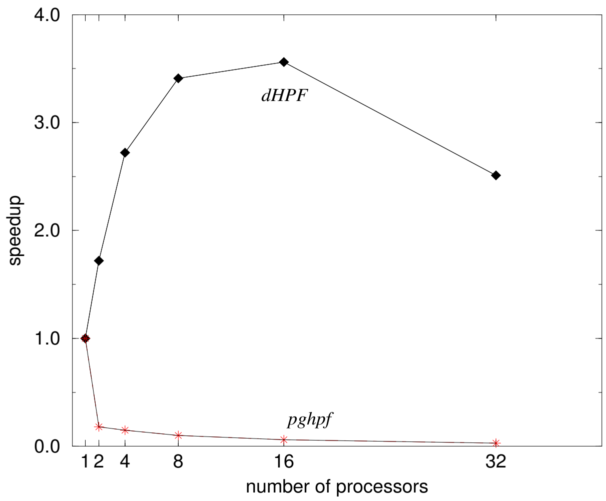

Figure 7. Speedups for Erlebacher reduction kernel benchmark.

4.3. Performance Impact of Factorization

As discussed in section 3.2, we use factorization to fragment one reduction statement into a group of reduction statements to better exploit data locality. Evidence that our factorization optimization is useful is supported by the presence of reductions that could benefit from factoring in benchmark programs such as wave5_data from SPEC95 and LWS from the PERFECT benchmarks to name a few.

To quantify the potential benefits from reduction factorization, we measure the execution performance of the synthetic benchmark shown in figure 8.

In figure 8, both a(n-j+1) and b(j) contribute to the reduction into `s`. The two array elements are distributed differently. Table 2 shows the performance of code generated by dHPF, *pghpf*, and *xlhpf* for the factorization benchmark with data size `n = 8192`. The dHPF compiler factors the

```
        program facTest
        integer n, w
        parameter (n=4096)
        real a(n), b(n)
        real s, t3
CHPF$   processors p(4)
CHPF$   template t(n)
CHPF$   align a(i) with t(i)
CHPF$   align b(i) with t(i)

CHPF$   distribute t(block) onto p

        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 8. Reduction factorization benchmark.

compiler	dHPF	pghpf	xlhpf
seconds	0.03	0.20	62.9

Table 2. Performance of code from three compilers on the factorization benchmark for `n = 8192`.

reduction so that the only communication necessary in the program is the single collective communication to compute the global accumulation for `s`. The *pghpf* compiler recognizes the reduction, but without factorization it communicates a vector of off-processor values for `a` before the `j` loop in each iteration of the timestep loop. The *xlhpf* compiler failed to recognize the reduction and inserted a broadcast of `t3` inside the `j` loop which resulted in poor overall performance.

To quantify the benefits of factorization in a more controlled experiment, we measure the performance of code generated by dHPF both with and without factorization. Without factoring support, statement `s = s + t3 + b(j)` gets the CP of ON_HOME `b(j)`. The statement `t3=a(n-j+1)` gets the CP ON_HOME `b(j)`, because, since `t3` is privatizable, dHPF propagates the CP from the use of `t3` to the assignment of `t3`. In this case, the code requires communication to get the off-processor values of `a(n-j+1)` to compute the assignment `t3=a(n-j+1)`. By performing reduction factorization, we split the reduction statement into a group of two statements: `s=s+t3` and `s=s+b(j)`. The statement `s=s+t3` gets the CP of ON_HOME `a(n-j+1)`, and `s=s+b(j)` gets the CP of ON_HOME `b(j)`. The accumulations for each of the resulting assignment statements are thus computed locally without the need for any communication.

n	4096	4096*4	4096*8	4096*16	4096*32
t_1	0.0379	0.0843	0.1489	0.2765	0.5250
t_2	0.0669	0.1883	0.3285	0.5933	1.0364
Δ	0.0290	0.1040	0.1796	0.3168	0.5114
o	0.765	1.219	1.205	1.146	0.974

Table 3. Execution performance with and without factorization (t_1 is the time for programs with factorization support, t_2 is the time for programs without factorization, $\Delta = t_2 - t_1$, and $o = \frac{\Delta}{t_1}$).

We measure the code generated by the dHPF compiler, with and without factorization support enabled, for different values of `n`. In table 3, the row labeled t_1 shows the execution time with factorization support; the row labeled t_2 shows the time without factorization support. The row labeled Δ shows $t_2 - t_1$: the extra communication time incurred without factorization support. The final row shows $o = \Delta/t_1$, the relative overhead of not using factorization support. In executions without factorization support, the execution times are roughly double because of the extra communication costs.

5. Conclusions

In this paper, we present effective reduction recognition and code generation techniques for compiling implicit reductions to distributed memory multiprocessors. This support has been implemented in the dHPF compiler. By using dependence analysis and pattern matching coupled with a flexible code generation framework, we handle a broad range of parallel reductions intermixed with other code in imperfectly nested loops that may or may not contain conditional control flow.

Minimizing communication is very important for generating efficient reduction code for distributed-memory machines. We describe techniques for reducing the number of collective communications: forming reduction groups, index encoding for MINLOC and MAXLOC reductions, and factorization to avoid unnecessary data movement. We compared the effectiveness of our techniques by benchmarking code generated by the dHPF compiler against code generated by IBM's *xlhpf* compiler and PGI's *pghpf* compilers. Our experiments show that our reduction recognition is effective and recognizes reductions in cases where one or more of *pghpf* or *xlhpf* does not. In addition, our communication optimization techniques for reductions lead to a factor of two or more performance improvement over code generated without them in the cases where they apply.

Finally, our optimizations to minimize communication can also be used to improve the code generated for distributed-memory machines using other recurrence parallelization techniques such as those of Fisher and Ghuloum [8, 7]. In addition, although we focus on generating code for a message-passing communication model in our exposition here, we are also using the same approach to optimize communication when generating reduction code for distributed shared memory systems as well.

6. Acknowledgments

The experiments were performed on a 64-processor IBM SP2 at the University of Houston. The Portland Group provided a copy of *pghpf 2.2* for benchmarking use. This work has been supported in part by DARPA Contract DABT63-92-C-0038 and sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency and Rome Laboratory or the U.S. Government.

References

[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, Apr. 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, Aug. 1996.

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

[5] 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, Oct. 1991.

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

[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, Dec. 1995.

[10] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

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

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

[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, Jan. 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] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[16] 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.