Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines

Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani
Tokyo Research Laboratory, IBM Japan
1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa 242 Japan
{suganuma, komatsu, nakatani}@trl.ibm.co.jp

Abstract

This paper presents a new technique for detecting and optimizing reduction operations for parallelizing compilers. The technique presented here can detect reduction constructs in general complex loops, parallelize the loops containing reduction constructs, and optimize communications for multiple reduction operations. The optimizations proposed here can be applied not only to individual reduction loops, but also to multiple loop nests throughout a program. The techniques have been implemented in our HPF compiler, and their effectiveness is evaluated on an IBM Scalable PowerParallel System SP2 using a set of standard benchmarking programs. Although the experimental results are still preliminary, it is shown that our techniques for detecting and optimizing reductions are effective on practical application programs.

1 Introduction

A considerable amount of research has been made on parallelizing compilers for both distributed-memory and shared-memory machines, and a number of techniques for analyzing loop parallelization and communication optimization have been proposed and implemented in those compilers [1,2,3,4,5,6]. However, reductions have not received significant attention as an important source of parallelism in general loops.

A reduction is an operation that computes a result, typically a scalar value, from a set of source arrays by reducing the number of dimensions on the basis of an associative binary operator. Reduction operations are very common in a broad range of numerical algorithms. Reductions are inherently sequential operations, and loops containing reduction operations generally do not fall into a model of parallelizable loops because of existence of the loop-carried data dependencies associated with reduction variables. In general, a technique using what is known as a parallel reduction algorithm [22] has been employed to execute this type of loop efficiently on a parallel machine. For a parallel machine with p processors and a target array of length n decomposed into segments of n/p, a reduction operation is executed in time of the order of $n/p + \log p$.

The conventional method of handling reductions [7,8,9,10, 11,12,13,14,15,16,17] relies on detection based on the idiom recognition of a given program fragment, and its transformation into a library call that is implemented for its fast and efficient solution on an underlying machine. Most of today's commercial compilers for parallel machines handle reduction loops in this way. This method however, has the following three limitations.

First of all, the detection patterns are limited by the pattern database, since the idiom recognition method defines a set of rigid syntactic patterns, over which it tries to match a given loop body. There are an unlimited number of reduction expressions, which cannot be covered in the compiler's database. Syntactic pattern-matching cannot detect reduction operations in arbitrary complex loops.

Secondly even if a reduction is detected in a loop, it is not always possible to transform the reduction loop into a library call, since reduction operations generally appear in an arbitrary nested loop with multiple conditionals and statements, which is not necessarily separated as distinct loops via loop distribution. For example, there could be a loop in which a reduction statement and another statement share the same reference variable defined in the same loop. The dependence cycle may exist and thus prevent the reduction statement from being separated.

Thirdly, the conventional method misses an opportunity for global optimizations to merge multiple reduction communications into a single event. The communication for a reduction combines partial results of local computation executed in parallel to obtain the final result, and it typically sends and receives a scalar data value in a packet of a communication event. All the processors have to be synchronized for gathering and scattering the values under a combining tree. This method of communication for reductions tends to reduce the performance of programs, and may become a serious performance bottleneck, especially when the number of processors is large. Since it is very common to execute multiple reduction operations in a single loop for practical applications, as in the case of computing a maximum array element and its index values, combining reduction communications is indeed a very important technique.

In this paper, we present a new technique for detecting and optimizing reduction operations for parallelizing compilers. The contributions of this paper include:

- *Technique for detecting reduction operations contained in a general complex loop.*
  This method is based on strongly connected components of the data dependence graph to extract candidates of reductions from the loop body, and analyzes the expressions of these candidates to determine whether they actually have reduction semantics. It can detect a broad class of reductions by finding a reduction operator and a reduction application function.

- *Parallelization of a general loop containing reductions among other statements.*
  This is done by means of a loop transformation, in which all the reduction variables are privatized. This method enables the loop to be parallelized by treating it as a local computation loop for the reduction statements.

ICS'96, Philadelphia, PA,USA
© 1996 ACM 0-89791-803-7/96/05..$3.50

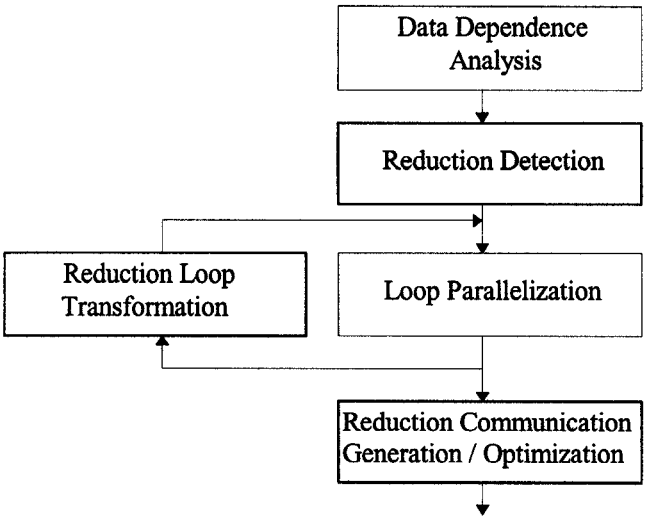


Figure 1: Diagram of our technique

- *Global communication optimization, achieved by coalescing and aggregating multiple communication events for reduction operations in a series of loops.*
  This optimization reduces the number of communication events associated with reduction operations, and significantly improves the performance of a program.

The rest of this paper is organized as follows. Section 2 describes an overview of our technique for detecting and optimizing reduction operations. Section 3 gives the proposed method of finding reductions in a general complex loop. In section 4, we describe our parallelization strategy for a loop containing reductions. Section 5 gives a new method for optimizing global communications for multiple reductions by coalescing and aggregating them into a single communication event throughout a program. Section 6 gives experimental results for some typical reduction loops and standard benchmarking programs. Section 7 describes related work, and section 8 concludes the paper.

2 Overview

The flow diagram of our technique is shown in Figure 1. We assume that the data dependence analysis have been done for a given loop prior to the reduction detection. Since a reduction inherently has a dependence cycle itself among statements forming the reduction, a strongly-connected component of the data dependence graph is considered to be a primary domain for the reduction detection analysis. Thus our reduction detection technique first extracts a set of statements as a reduction target in a complex loop body, by focusing on the strongly connected component, and then it recognizes whether the set of statements actually form a reduction by finding a reduction operator and a reduction application function.

The loop transformation for a reduction is applied when the parallelization fails for the loop containing reductions. The transformation is done in such a way that all the reduction variables are privatized. This eliminates the loop carried data dependence associated with reduction variables. If parallelization succeeds in the second trial, the loop can be treated as local computation loop for reduction operations, and the communications for reductions are generated to combine the partial results. Otherwise, the reduction variables can be used as final results wihtout causing communication for combining. Finally the communication optimization for reductions is applied within an entire program to coalesce and aggregate multiple reduction communications to a single event throughout the program.

3 Reduction Detection

Our reduction detection algorithm is based on the analysis of equations constructed from a set of statements using information in the data dependence graph (DDG). The algorithm consists of the following three passes for each loop in a given program.

- Scalar privatization analysis
- Reduction target extraction
- Reduction recognition

Reductions can be expressed in the form of the following quintuple:

$$R = \{v, b, f, g, D\},$$

where v represents a reduction variable, b represents an associative reduction operator, f is a function to be applied in executing a reduction, g is a predicate of a logical expression that specifies the guard to the reduction execution, and D is a domain for executing the reduction operation. Unless f is an identity function, the reduction intrinsically requires multiple values to be combined under a combining tree, one to lead the final result, and others to apply the function f in the combining phase.

The detection algorithm automatically constructs the quintuple from a set of statements in a loop if they have reduction semantics. Examples show the process to form the reduction quintuple for two cases.

3.1 Scalar Privatization Analysis

The first pass of the algorithm performs the scalar privatization analysis [5]. It finds those scalar variables whose definitions and references are closed in a loop iteration. Namely for each of the references of a scalar variable in a loop, all the reaching definitions have to be done in the same loop nest level. The result of the analysis is used in the algorithm in that, non-privatization is a feature of reduction variable candidates, and privatized scalar variables can be terminated points to construct equations in the pass of the reduction target extraction.

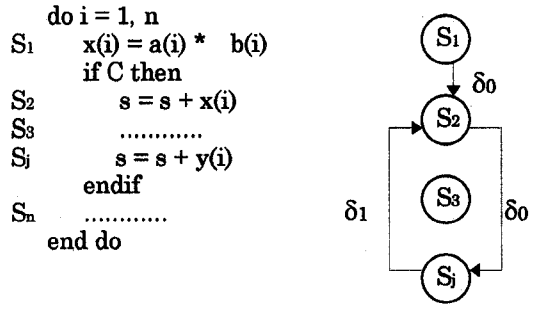
3.2 Reduction Target Extraction

The second pass of the algorithm extracts targets for recognizing a reduction from a loop body. It further consists of two steps. The first step builds an equation for each of the assignment statements in a given loop body. Let e be the equation for a statement S; then e can be expressed by

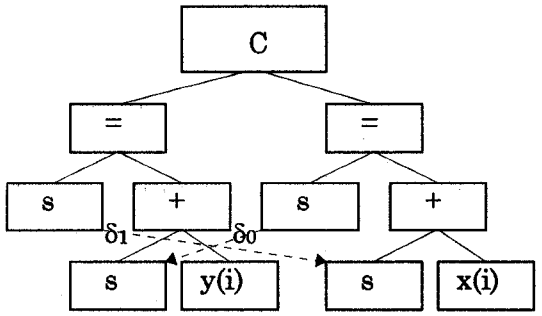
$$e(S) = \{z, E, P\},$$

where z is a variable of the left-hand-side of the statement, $E = (e_1, e_2, \ldots e_m)$ is an expression of the right-hand-side of the statement consisting of variables $e_1, e_2, \ldots e_m$, and $P = (p_1, p_2, \ldots p_n)$ is a set of predicates $p_1, p_2, \ldots p_n$ of logical expressions that guard the execution of the statement S. The expression is represented as an expression tree.

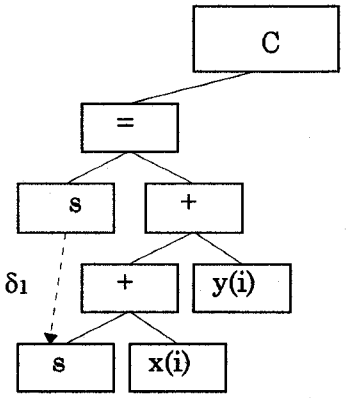
The second step merges the equations of assignment statements by forward substitution based on the strongly-connected component (SCC) of the data dependence edges associated with non-privatized scalar variables, if those statements share the same conditional expressions. By using the SCC, statements relevant to a reduction can be extracted from a general loop body. Two equations



(a) Loop with conditional statements and its SCC



(b) Expression trees of the loop body statements



(c) Expression tree after a merge operation

Figure 2: Reduction loop of conditioned SUM

$$e(S_i) = \{z_i, E_i, P_i\}, E_i = (e_{i1}, e_{i2}, \cdots, e_{im_i})$$
$$e(S_j) = \{z_j, E_j, P_j\}, E_j = (e_{j1}, e_{j2}, \cdots, e_{jm_j})$$

can be merged into

$$e(S_l) = \{z_l, E_l, P_l\},$$

if two predicates are identical, *i.e.* $P_i = P_j$ holds and there is a non-privatized scalar variable $e_{jk} \in E_j, (1 \le k \le m_j)$ such that z_i and e_{jk} are instances of the same variables and a loop independent data dependence [23] exists from z_i to e_{jk}. The merged expression E_l should be the expression E_j in which all references to the variable e_{jk} are simultaneously replaced by the respective expression E_i. The left-hand-side variable z and the predicates P of the merged equation must be $z_l = z_j$ and $P_l = P_j$, respectively.

3.3 Reduction Recognition

The final pass of the algorithm determines the existence of reduction operations by analyzing the equations generated in the previous steps. An equation $e = \{ z, E, P \}$, where $E = (e_1, e_2, \ldots e_m)$ and $P = (p_1, p_2, \ldots p_n)$, derived by previous steps, can be recognized as a reduction if either of the following conditions are satisfied. The first condition examines the structure of the expression E, when the left-hand-side variable z can be found in E. Namely, when there exists a variable $e_j \in E, (1 \le j \le m)$ such that $e_j = z$, and a series of operations involving the operand e_j in the expression are all equal commutative and associative operator b, then the equation is recognized as a reduction

$$R = \{z, b, I, P, D\},$$

where I is an identity function, and D is the iteration space executed by the enclosing loops. This condition ensures that the reduction function being recognized is actually closed under the composition operation.

The sample program loop Figure 2a consists of the reduction of two statements S_2 and S_j with a mask expression, among other statements. As shown in the figure, S_2 and S_j consist of a strongly connected component in the data dependence graph by the edges associated with the variable s. The equations generated in the first half of the second pass in the algorithm are indicated in Figure 2b, where the conditional statement with the predicate C is pointed to by the expression trees of the body statements to reflect the control dependence. Thus the expressions of nested conditionals can be reached from its body statements by traversing the connections from the expression trees. Figure 2c shows the merged expression tree derived by the second pass of the algorithm, using the strongly connected component by data dependence edges associated with the non-privatized scalar s. The merge of two equations in Figure 2b has been done in such a way that it eliminates the loop-independent data dependence δ_0 between the variable s, which are shared by two equations. The final step of the algorithm determines that this expression actually constitutes a reduction. The traversal of the expression tree from the reduction variable up to the root reveals that it has the same binary operator '+', which is commutative and associative, resulting in the reduction of

$$R = \{s, SUM, I, C, D\},$$

where I is an identity function, and D is the iteration space of the enclosing loop.

The second condition examines whether one of the predicates in P and the expression E form the reduction semantics. If there exists a predicate $p_i = P, (1 \le i \le n)$ such that p_i has a logical expression with a relational operator *relop*, and the expression has the form of

$$f(z)\ relop\ f(E)$$

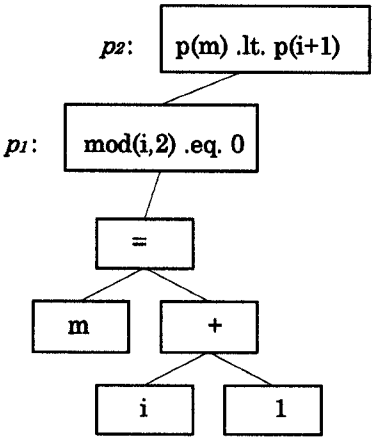
for a pure function f, then the equation is recognized as a reduction with the form

$$R = \{z, b, f, P, D\},$$

where the operator b is determined by the relational operator *relop* in the predicate p_i, and D is the iteration space of the loop enclosing the reduction statements. The function f can be an arbitrary pure function in terms of the language specification of High Performance Fortran [19], that is, it has no side effect for the function application. Examples of such functions are elementary intrinsic functions such as *f(x) = abs(x)*, and an array element referencing function *f(x) = A[x]* for an array A.

```
do i = 1, n - 1
   if (p(m) .lt. p(i+1)) then
      if (mod(i,2) .eq. 0) then
         m = i + 1
      endif
   endif
enddo
```

(a) Loop with nested conditional statements



(b) Expression tree of the loop body statement

Figure 3: Reduction loop of conditioned MAXLOC

The example in this case is the loop locating the index value of the maximum element of the array, which is guarded by a mask statement, as shown in Figure 3a. This is a slightly modified loop of the Livermore kernel 24. Figure 3b shows the expression tree generated after the second step of the algorithm for this loop. Since it does not find the reduction variable in the right-hand-side expression, the reduction determination pass in this case refers to the guard expressions, searching for a predicate in order to decide whether to form a reduction operation together with the logical expression of the predicate. The examination of the predicates reveals that p_2 has the *relop* of '<' and has the logical expression of

$$f(m)\ relop\ f(i+1)$$

for a pure function $f(x) = p[x]$, which returns an element of the array p. Thus it can be recognized that the statements are the reduction of

$$R = \{m, MAX, f, p_1 \wedge p_2, D\},$$

where D is the iteration space of the enclosing loop. This reduction has the semantics of MAXLOC from the relational operation '<' and the array element referencing function f.

4 Reduction Loop Transformation

A reduction construct detected by the above algorithm generally consists of several statements in an arbitrary nested, imperfect loop, and it contains several target arrays with different data decompositions over processors. Reductions of this type cannot be handled by the conventional method, that is by the direct transformation of the reduction statement into the corresponding library call. In contrast, our basic approach keeps the loop formation of the reduction as a local computation loop, and it relies on the loop analyzer for its parallelization. This approach enables a loop containing reduction constructs to be parallelized, if the loop is appropriately modified to a form amenable to the loop analyzer.

Reduction Computation Partitioning

All the reduction variables are privatized as the first step. This is done by duplicating the reduction variables that are private to each processor, and replacing original variables in reduction statements with the local reduction variables. It is possible to privatize them by taking advantage of the associative property of reduction operations. Then the reduction computation partitioning is applied to reduce the amount of communication based on the data decomposition. This results in the partitioning of the computation domain D in the reduction. Here the result of the communication analysis is used to check the validity of this transformation.

Communication analysis [2,4] is a set of techniques employed in parallelizing compilers for distributed-memory machines to identify the flow of data among processors, based on the data dependences and the data decomposition for each operand. It determines whether a given loop is parallelizable and where the communication calls should be inserted in the nested loops. For our purpose, it determines the reduction operands that can be prefetched.

A new loop is generated if there are subexpressions in the reduction computation, which are denoted as node t in the reduction expression tree, that satisfy the following conditions: (1) all the operands under the subtree rooted at the node t have been identified by communication analysis as prefetchable at the same nest level of the enclosing loops; (2) all operators involving the subexpression are the same as the reduction operator. The new loop is then generated at the prefetchable nest level with a reduction statement as its body, which is constructed by combining these subexpressions under the reduction operator. The generated loop can be parallelized by privatizing the reduction variable. This transformation can be considered as an optimization for reducing the data prefetching communication overhead for a parallelized loop, achieved by exchanging a single value of the local reduction result instead of all the elements of a required section of the operands. The neutral element of the privatized reduction variable can be determined by the reduction operator b and the reduction application function f.

Communication Generation

The loop transformation described above allows reduction loops to be successfully parallelized, unless the loops are inherently sequential for some reasons other than the presence of reduction statements. If the loop is successfully parallelized, the loop is treated as local computation loop for the reduction, and the global communication is generated for combining the reduction partial results.

For the reduction

$$R = \{z, relop, f, P, D\},$$

where the processor i executing the domain D_i produces the partial result z_i, the pair of the partial values $(z_i, f(z_i))$ has to be passed to the communication library. The value z_i is to lead the final result, and the value $f(z_i)$ to apply the reduction operator *relop*. The combining operation for $(z_i, f(z_i))$ and $(z_j, f(z_j))$ from processor i and j respectively can then be described as follows, where 'then' and 'else' parts have to be actually determined by the operator *relop*.

$$if\,(relop(f(z_i), f(z_j)))\,then\,(z_i, f(z_i))$$
$$else\,(z_j, f(z_j))$$

5 Reduction Coalescing and Aggregation

The global communication of integrating intermediate results into the final result has to be applied for a parallelized reduction loop. In general the communication is generated for each of the reduction operations in a whole program, and it synchronizes all the related processors. For multiple reduction operations in a single loop or across different loop nests, the reduction communications can be merged into a single communication event by either coalescing or aggregating those communications, if there are no intervening references to those reduction variables. This amortizes the high message passing overhead of the collective communication. It is very common in practical numerical programs to perform multiple reduction operations in a single loop body, as in the case of computing the maximum and minimum values of an array together with the array subscript values.

The reduction communication typically consists of three steps:

- Determining the group of participating processors for the communication from the domain of the reduction.
- Gathering the intermediate results into a processor from the processor group under a given reduction operator.
- Scattering the final result to the processors in the group.

Since the computation domains are not necessarily identical for each of reductions, the communications have to be merged for a larger set of processors, which can be derived from the union of the computation domains associated with each reduction operation.

Let an intermediate state of a reduction be denoted by $Q = \{v, b, f, D\}$ after the local computation for the parallelized reduction $R = \{v, b, f, g, D\}$. The guard expression g in the reduction R is dropped, since it is used only for the local computation part. The reduction coalescing and reduction aggregation can then be described as follows.

Reduction Coalescing

The reduction coalescing merges a set of reduction communications sharing the same reduction variable and the same combining operator. The intermediate states of two reductions

$$Q_i = \{v_i, b_i, f_i, D_i\} \text{ and}$$
$$Q_j = \{v_j, b_j, f_j, D_j\}$$

can be coalesced into

$$Q = \{v_i, b_i, f_i, D_i \cup D_j\},$$

if $v_i = v_j$, $b_i = b_j$, and $f_i = f_j$ hold, and there is no reference to the variable v_i in the execution flow between the reductions Q_i and Q_j in the program. This avoids the execution of unnecessary collective communications that do not lead to the final result.

Reduction Aggregation

While the reduction coalescing ensures that each variable is communicated only once under a reduction operator, the reduction aggregation ensures that only one communication is performed for a set of communications required by the reduction variables. The intermediate states of two reductions

$$Q_i = \{v_i, b_i, f_i, D_i\} \text{ and}$$
$$Q_j = \{v_j, b_j, f_j, D_j\}$$

can be aggregated into

$$Q = \{(v_i, v_j), (b_i, b_j), (f_i, f_j), D_i \cup D_j\},$$

where (v_i, v_j), (b_i, b_j), and (f_i, f_j) are vectorized reduction variables, operators, and functions respectively, if $v_i \neq v_j$ holds and there is no reference to the variable v_i between the reductions Q_i and Q_j in the program, assuming that Q_i has precedence over Q_j in program execution. The aggregated reduction executes a reduction operation to each of the vectorized reduction variables using the corresponding combining operator. Here a processor guard has to be provided for each of the reductions to ensure that the reduction communication is performed by a correct set of processors. For this purpose, each domain for reduction computation is also vectorized and passed to the aggregated reduction. In other words, given a set of reduction variables, combining operators, and reduction domains, each processor first determines whether it should participate in the reduction operation. If it should, it applies the specified combining operator to the corresponding input variables; otherwise, it just passes the input variable to the output.

```
asum = SUM (a(1:n)
amax = MAXVAL (a(1:n))
```

(a) Consecutive call of F90 reduction intrinsics

```
#1 = 0
do i = 1, n
  #1 = #1 + a(i)
end do
#2 = smallest_val
do i = 1, n
  #2 = max (#2, a(i))
end do
call reduce ((#1,#2),(SUM,MAXVAL),
             (I,I),(1:n))
asum = #1
amax = #2
```

(b) Inlined reduction loops and the aggregation

Figure 4: A series of F90 reduction intrinsics and its aggregation

Figure 4a shows an example of a series of reduction statements described by F90 intrinsic functions. The reduction aggregation can be effectively applied even in this case after each of the intrinsic statements has been inlined into a reduction loop as shown in Figure 4b. The reduction temporary variables, #1 and #2, the combining operators, SUM and MAXVAL, and the identity functions, *I*, are vectorized, respectively. Since the computation domains of the reductions are the same, that is $D_1 = D_2 = D_1 \cup D_2$, processors guard is not required in this case.

Since the reduction communication typically sends and receives at most a single value at a time in a communication event, the global optimization results in an increase of the packet utilization rate for a reduction communication and a decrease of the number of communication events and synchronization points, and thus helping to improve the performance of source programs. The communication overhead reduced by this optimization can be described as follows. The cost of executing a reduction communication among p processors is described as $C_1 \log p + C_2$ where C_1 and C_2 are both constants. So if we let n and m be the numbers of processors for two reductions, and p be the number of processors of a reduction merging them, then

```
do j = 1, n
do i = 1, m
  if (gmax.lt.grid(i,j,iold)) gmax = grid(i,j,iold)
  if (gmin.gt.grid(i,j,iold)) gmin = grid(i,j,iold)
end do
end do
```

Figure 5: Reduction loop in the grid benchmark

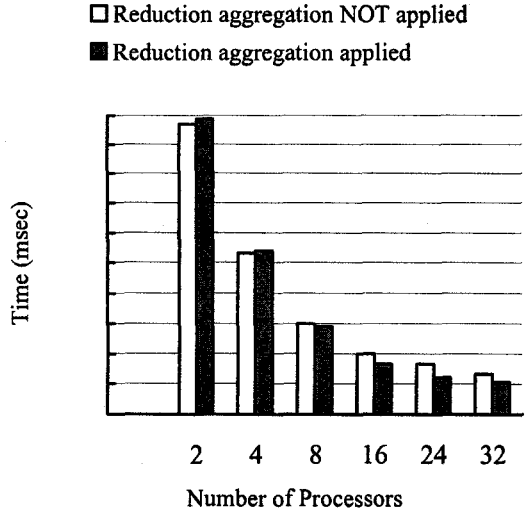


Figure 6: Execution time of the grid reduction

$$C_1 \log p + C_2 \le C_1 \log(n+m) + C_2 \le C_1 \log(nm) + C_2 \le C_1 \log n + C_1 \log m + C_2,$$

since $p \le n+m$. Practically the decomposition specification of reduction target arrays can be considered to be equal in most cases. This means that the number of processors involved in multiple reductions are equal; that is, $p = n = m$. This halves the communication cost of reductions.

6 Experimental Results

We have implemented the above techniques for handling reductions in our HPF compiler [4,18]. The compiler performs various optimizations, including loop parallelization, communication optimizations, and locality analysis.

To evaluate the effectiveness of our techniques on distributed-memory machines, we performed two kinds of experiments: one for focusing on reduction loops thenselves to check the benefit of the reduction communication optimizations, and the other for examining the speedup of standard benchmarking programs to evaluate how our techniques affect the total performance of practical programs. The detection and optimization for reductions used in these evaluations have all been implemented on and are applied automatically by our HPF compiler. All the experiments described here were conducted on an IBM Scalable PowerParallel System SP2.

6.1 Reduction Loop Evaluation

We first examine the reduction loop extracted from the grid benchmark shown in Figure 5. This example of computing an array maximum and minimum value in the same loop body is considered to be a typical problem that frequently appears in practical application programs. The optimization of reduction aggregation can be used to merge two reduction communications. Figure 6 shows the time associated with the execution of the reductions when the loop is iterated 100 times. The time includes the reduction local computation, communication, and initialization of reduction variables for each iteration. The data size is 500 x 500 x 2, that is, n = m = 500. As the figure shows, the effect of reduction aggregation increases with the number of processors, reaching a gain of about 24% over the non-aggregated version with 32 processors. When the number of processors is small, the result of the reduction aggregation in this example is slightly worse than that of the non-aggregated version. This is because the overhead for the aggregated reduction communication, due to the processor guard execution, is not fully covered by the benefit of reducing the number of communication events with the small number of processors.

```
      do 270 j = 1, m
      do 270 i = 11p, 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 7: Reduction loop in the tomcatv benchmark.

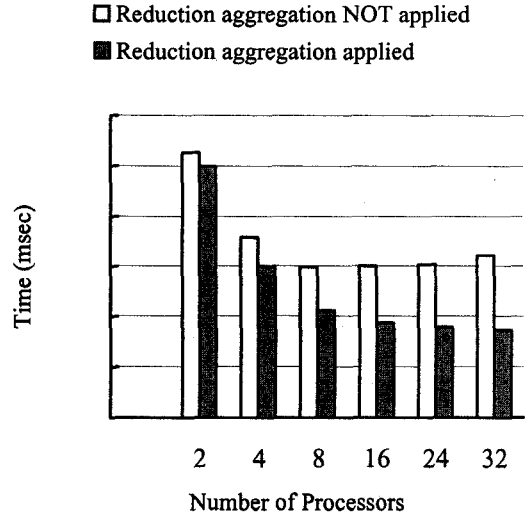


Figure 8: Execution time of tomcatv reduction loop

We now look at the next reduction loop extracted from tomcatv, a mesh generation program in the SPEC92 floating-point benchmarking suites. This reduction loop shown in Figure 7 computes six reduction variables: two for computing the maximum absolute values of arrays and four for computing the array subscripts values at that time. Reduc-

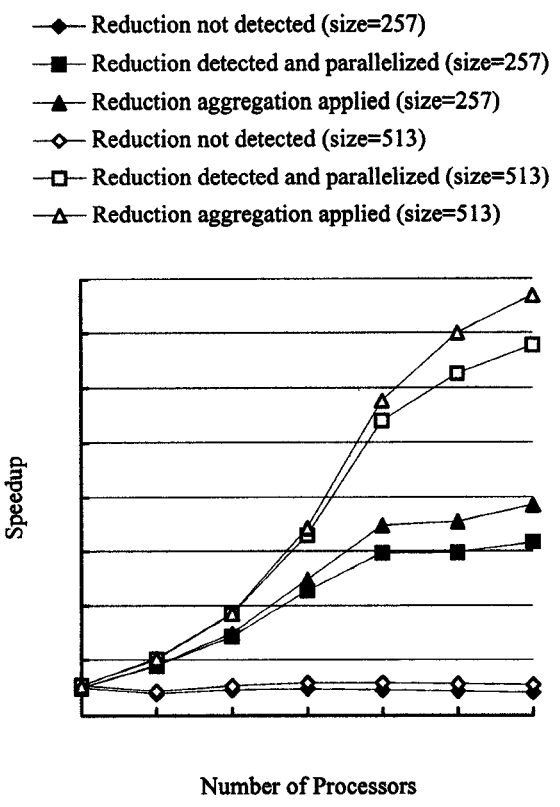


Figure 9: Speedup ratio of the tomcatv benchmark

tion aggregation is again used in this case to combine these reduction communications. Figure 8 shows the time of this reduction loop iterated 100 times. Here the data size is 257 x 257, the same as in the tomcatv program. Since the data size is not large enough, the performance of non-aggregated reduction becomes saturated with about eight processors, and it begins to decline thereafter. In contrast, the aggregated version shows a performance gain with large numbers of processors. The effect of the reduction aggregation is greater than that in the previous example, since it aggregates six reductions into a single communication event. The gain rises to 47% over non-aggregated version with 32 processors.

6.2 Benchmark Evaluation

We now examine the performance of tomcatv to demonstrate how the reduction detection and optimization techniques affect the total performance of the whole program. The reduction loop, shown in Figure 7, is enclosed by the outer iterative cycle, whose results are used for checking the convergence behavior of the program. The performance data on parallel machines for this benchmark reported in various publications are based on the modified version of the program, in which the statements computing subscript values are eliminated, and typically the Fortran90 intrinsic function of MAXVAL is used to replace the statements for computing the maximum absolute values of arrays. In contrast, our benchmark result presented here is based on the original program with annotation of only HPF directives for specifying data decomposition.

Figure 9 shows the set of speedup curves over the best sequential version of the program, for different data sizes: 257 x 257 and 513 x 513. Here the decomposition of all the array data is specified as (block,*), and the program converges after 100 iterations. The figure shows that when the program fails to detect reductions, its performance does not scale. This is because the overhead of the serial execution of reductions kills the gain produced by parallel computations.

The reduction aggregation is effective in this program, since the reductions are used in the iterative algorithm to check the convergence behavior for approximate solutions. The reason for the slowdown in the speedup curve for the case of more than 16 processors is considered to be that SP2 nodes are grouped into 16-processor units connected to switching boards for the interconnection network, and thus the communication via an external link becomes expensive with processors larger than 16.

Overall, the optimization of aggregating multiple reduction communications contributes around 10%-25% of the speedup over the non-optimized version when the number of processors is greater than eight. This tends to become larger as the number of processors increases.

7 Related Work

Our global optimization technique for multiple reduction operations over a whole program is unique and there has been no comparable approach for executing reductions efficiently. We therefore restrict our dicussion here to the papers dealing with the detection problems. Detection and efficient execution of scan and reduction operations in a program have been extensively studied, mainly in the field of vectorizing compilers. Most of these work focuses on detection by syntactic pattern-matching of a loop body.

Pinter and Pinter's method [8] is to construct what they call a computation graph by PDG-based analysis and then rewrite the graph by using a list of idioms. Their method can handle some conditionals for recurrence operations; however, it essentially matches individual patterns in the computation graph by reference to a pattern database of a set of structures, and it replaces them using the graph rewrite rules. In contrast, our method by analyzing expression trees can find a broader class of reductions, which is enabled by finding a reduction application function in addition to a reduction operator.

Radon and Feautrier [9,10] proposed a semantic technique using algebraic properties to simplify complex loop body structures; however, they eventually rely on pattern-matching for recurrence operators.

Callahan's method [11] is to model bounded recurrent loops as functions with the properties closed under composition and it provides an algebraic approach to recognize recurrences; in the end, though, it matches patterns by using a set of core recurrent operations.

Fisher and Ghuloum [12,13] proposed a new technique for finding scans and reductions by extracting a loop modeling function, and then reassociating it with a series of functional compositions. Their method is powerful and unique in that it relies on the analytical ability rather than on a database to find recurrences. However, it implicitly assumes that reduction loops are encoded separately by programmers or isolated by loop distribution. In contrast, our method detects reductions from complex loops with multiple conditionals and statements where reductions are hidden deeply in those loop nests, and it parallelizes the loops by privatizing reduction variables.

Pottenger and Eigenmann [16] described a method of reduction recognition, which is capable of handling histogram reduction and single address reduction. It transforms loops into three types of parallel reduction loops: blocked, privatized and expanded. In contrast to our method, their method deals only with separated and isolated reduction loops.

8 Conclusion

We have presented a new technique for detecting and optimizing re-

duction operations. It covers the detection of reduction statements that are contained in a general complex loop, which can find a broad class of reductions by allowing a reduction application functions. It parallelizes general reduction loops and optimizes communications globally by coalescing and aggregating multiple reduction communications within an entire program. The techniques presented here have been implemented in our HPF compiler, and their effectiveness has been evaluated for some reduction loops and a set of benchmarking programs. The experimental results have shown that our techniques are effective for practical application programs, especially when the reduction is used to check the program convergence behavior in iterative algorithms. This form of reductions seem to appear frequently in those numerical algorithms which approximate solutions for linear systems of equations.

The discussion, the implementation, and the evaluation of our technique have been based on a distributed-memory machine in this paper; however, the technique is also applicable for shared-memory machines. The differences between them lie in the reduction variable privatization and the code generation for combining intermediate results. A private copy of a reduction variable, which is shared, is used in the parallel loop, and locks have to be used for combining the partial results. Reduction coalescing and reduction aggregation can be considered beneficial even for shared-memory machines.

In this paper, we have focused on the detection and the global optimization of reduction statements; however, our framework can be considered extendable for linear recurrences, such as prefix operations, beyond reductions. The main difficulty with handling recurrence operations resides in that special care has to be taken with the sequence of combining intermediate results in the transformation and optimization phases. Segmented scans and reductions [20] have the similar problems. The formal extension of our framework to these recurrence operations will be the focus of our future work.

Acknowledgements

We thank all the members of the Advanced Compiler group of IBM Tokyo Research Laboratory for their helpful discussions and comments on this research.

References

[1] H. Zima and B. Chapman. Compiling for distributed-memory systems. In *Proceedings of the IEEE*, vol. 81, No. 2, Feburuary 1993.

[2] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, Vol. 35, No. 8, pages 66-80, August 1992.

[3] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 1-13, Albuquerque, NM, June 1993.

[4] K. Ishizaki and H. Komatsu. Loop parallelization algorithm for HPF compiler. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[5] D. E. Mayden, S. P. Amarasinghe and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proceeedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 2-15, Charleston, SC, January 1993.

[6] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K.Y. Wang, and M. Burke. PTRAN II: A compiler for High Performance Fortran. In *Proceedings of Fourth Workshop on Compilers for Parallel Computers*, Delft, Nethwelands, December 1993.

[7] Z. Ammarguellat and W. L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 283-295, White Plains, NY, June 1990.

[8] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *Conference Record of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 79-92, Orlando, FL, January 1991.

[9] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *Proceedings of the Fifth International Conference on Parallel Architectures and Languages Europe (PARLE)*, pages 132-145, Munich, June 1993.

[10] X. Redon and P. Feautrier. Scheduling reductions. In *Proceedings of ACM SIGARCH International Conference on Supercomputing*, pages 117-125, Manchester, July 1994.

[11] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[12] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 135-146, Orlando, FL, June 1994.

[13] A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 58-67, Santa Clara, CA, July 1995.

[14] P. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. Do & Merge: Integrating parallel loops and reductions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[15] P. Jouvelot and B. Dehbonei. A unified semantic approach for vectorization and parallelization of generalized reductions. In *Proceedings of ACM SIGARCH International Conference on Supercomputing*, pages 186-194, Crete, June 1989.

[16] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings* of *ACM SIGARCH International Conference on Supercomputing*, pages 444-448, Barcelona, July 1995.

[17] P. Tang and N. Gao. Vectorization beyond data dependencies. In *Proceedings* of *ACM SIGARCH International Conference on Supercomputing*, pages 434-443, Barcelona, July 1995.

[18] T. Nakatani. Compiling HPF for a cluster of workstations. In *Proceedings of the Joint Symposium of Parallel Processing*, pages 1-6, Tokyo, May 1993.

[19] High Performance Fortran Forum. High Performace Fortran Language Specification. Version 1.0, May 1993.

[20] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, Vol. C-38, No. 11, pages 1526-1538, November 1989.

[21] W. D. Hillis and G. L. Steele Jr.. Data parallel algorithms. *Communications of the ACM*. Vol 29, No. 12, December 1986.

[22] M. Wolfe. *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989

[23] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, NY, 1991

[24] G. E. Blelloch. *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, MA, 1989