還原計算之最佳化及平行化
An Interleaving Transformation for Optimizing Reductions

吳真貞
Jan-Jan Wu
中研院資訊科學所
Institute of Information Science
Academia Sinica

摘要

本論文描述一可將還原計算(reduction)平行化之方法,稱為交錯轉換. 此方法結合資料從屬分析與區域分析,可抽取複雜之還原計算中之平行度. 本論文並以在二個平行機器上所得之實驗數據來驗證此方法.

Abstract

In this paper, we present a general compilation approach, *interleaving transformation*, for parallelizing reductions in loops. This optimization exploits parallelism embodied in reduction loops through combination of *data dependence analysis* and *region analysis*. We will show how this optimization extracts partial parallelism from reduction loops which contain data dependences. Experimental results on the Connection Machines CM-5 and the nCUBE2 are reported.

Keywords: reduction, parallelizing compiler optimization

1 Introduction

A reduction occurs when a location is updated on each loop iteration with the result of a commutative and associative operation applied to its previous contents and some data value. The simplest example is SUM, defined by $SUM(X) = \sum_{i=1}^{n} X(i)$. Strict interpretation of explicit loops for reductions such as

```
DO I=1,n
   sum = sum + X(I)
END DO
```

presents a problem to compilers, because dependences inherent in the straightforward implementation prevent parallelization of the loop. However, a loop containing a reduction may be safely parallelized since the ordering of the commutative updates need not be preserved. For example, to compute SUM of n values using p processors, one may divide the n values into p chunks, assign one chunk to a processor, and let every processor computes locally the sum of the values assigned to it, independent of other processors, and then, at the end, combine the partial results produced by these processors.

Parallelization of reductions require language constructs or compiler techniques that can help break data dependences in reductions. One obvious compilation approach is pattern recognition. Either the source language includes explicit reduction operators (e.g. the SUM operator in High Performance Fortran), or certain specific loops are recognized as equivalent to known reductions (e.g. the loop for SUM reduction as described above). Once such patterns are recognized, hand optimized code for the reductions are emitted in the code generation phase.

Previous work on pattern recognition for reductions had been reported in the Parafrase system [4], the Eave [1] system, and the Fortran D system [7]. Redon and Feautrier proposed an algebraic specification method for recurrences detection [6]. Pinter and Pinter proposed a matching method based on Program Dependence Graphs [5]. Fisher and Ghuloum reported a method for extracting recurrences from loop structures that contain conditional statements [3].

All these existing work only parallelize reduction loops that contain no stores that can overlap any loads within the loop (i.e. loops that only contain dependences caused by the reduction statement). For instance, the SUM loop given above can be parallelized straightforwardly by existing techniques. We call this class of reductions *basic* reductions. Not all reductions are *basic*, however. Consider the loop nest in `Program 1`, which solves a triangular linear system:

A reduction `B(J)=B(J)-L(J,I)*x(I)` is carried over the outer loop indexed by `I`. Two kinds of data dependences exist in the outer loop: a loop-carried dependence caused by the reduction statement, and a loop-carried dependence caused by computing the value of `x(I)`, whose value depends on the value of array `B` computed in previous iterations. Existing techniques fail to parallelize this reduction due to the second kind of dependence.

Program 1

```
REAL L(n,n), B(n), x(n)
DO I = 1, n
   x(I) = B(I) / L(I,I)
   DO J = I+1, n
      B(J) = B(J) - L(J,I) * x(I)
   END DO
END DO
```

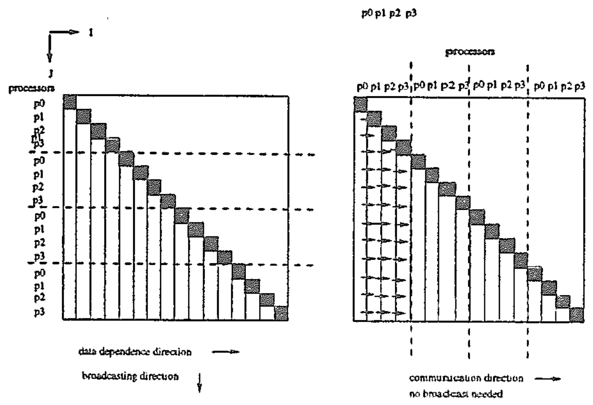
In this paper, we present a more general approach that is able to parallelize a broader class of reduction loops. The basic idea in this approach is to exploit partial parallelism embodied in reduction loops through combination of *data dependence analysis* and *region analysis*. Data dependence analysis identifies loop structures that contain reduction operations and the condition that can trigger this optimizing transformation. Region analysis extracts partial parallelism by separating reduction iterations into a sequential region and an order-insensitive region. Parallelism is achieved by interleaving reduction iterations in the order-insensitive region onto multiple processors. We will use the triangular linear-system solver as a running example to present the sequence of transformations that lead to an optimized parallel implementation. Experiments demonstrating the effectiveness of this optimization were conducted on the Connection Machines CM-5 and the nCUBE2.

2 A Motivating Example

We use the lower triangular solver L*x=B as shown by **Program 1** to motivate the optimizations. For the purpose of exposition, we expand array B with an additional time index. Let $b(J, I)$ denote the reduction result at time step I. Initially, $\forall J, b(J, 0) = B(J)$, and $b(J,I)=b(J,I-1)-L(J,I)*x(I)$.

2.1 Straightforward Implementations

We assume SPMD (Single Program Multiple Data) model for parallel implementation. One obvious parallelization strategy for Program 1 is to distribute array X and array L at the first dimension. Since the loops iterate over a triangular domain, cyclic data distribution is a reasonable choice for load balancing. Figure 1(a) shows the program running on each processor, where each processor is assigned n/p rows of array L and n/p elements of arrays B and X. Index I denotes the local index on each processor, and global_I denotes the global index converted from a local index. This approach achieves parallelism, however, at the expense of communication overhead for



(a) distribution at the first dimension (shaded area indicates array X)

```
REAL L(n/p,n), B(n/p), x(n/p)
DO I = 1,n/p
  global_I = I*p+my_pid
  x(I)=B(I)/L(I,global_I)
  broadcast x(I) to all processors
  DO J = I+1, n/p
    B(J) = B(J)-L(J,global_I)*x(I)
  END DO J
END DO I
```

(b) distribution at the second dimension no optimization of reductions

```
REAL  L(n,n/p), B(n), x(n/p)
DO I = 1, n/p
  global_I = I*p+my_pid
  recv B(global_I:n) from left
  x(I)=B(global_I)/L(global_I,I)
  DO J = global_I+1, n
    B(J) = B(J)-L(J,I)*x(I)
  END DO J
  send B(global_I+1:n) to right
END DO I
```

Figure 1: Straightforward Parallel Implementations of the Triangular Solver

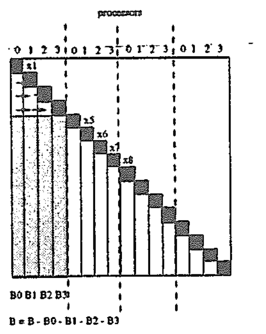
broadcasting data elements of X in each outer loop iteration. Parallelization of reductions is not possible since all the outerloop iterations are assigned to a single processor.

An alternative approach is to distribute L at the second dimension. The reduction terms are now distributed among multiple processors. Without parallelization, the computation for reductions is sequentialized and communications between adjacent processors are required for each outer loop iteration, as shown in Figure 1(b).

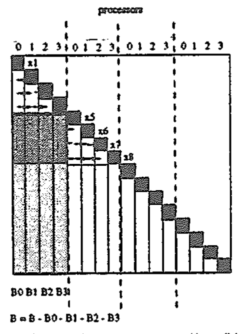
2.2 Optimized Implementations

Performance of the program in Figure 1(b) can be further improved by interleaving the reduction terms as shown in Figure 2(a). For each outer iteration, the reductions in the gray area can be computed in parallel. The partial results must be combined before entering next outer loop iteration. Communication can be further reduced by combing the partial results in the dark gray area only, and postpone the others to the next outer loop iterations, as shown in Figure 2(b).

This example gives intuition of interleaved reductions. Formalization of the transformation follows. We first define the input loop forms to the transformer. We then describe the associated analysis technique and the program transformations.



(a) reductions in the gray area are computed in parallel and combined at the end of the computation



(b) reductions in the gray area are computed in parallel and only the partial results in the darker gray area are combined at the en

```
REAL  L(n,n/p), B(n), x(n/p)
DO I = 1, n/p
  global_I = I*p + my_pid
!!! sequential between processor
  recv B(global_I:I*p) from left
  x(I)=B(global_I)/L(global_I,I)
  DO J = global_I+1, I*p
     B(J) = B(J)-L(J,I)*x(I)
  END DO J
  send B(global_I+1:I*p) to righ
!!! parallel between processors
  DO J = I*p+1, n
     B(J) = B(J)-L(J,I)*x(I)
  END DO J
!!! combine partial results in
!!! the gray area
  global_sum(B(I*p+1:n),1:p)
END DO I
```

```
REAL  L(n,n/p), B(n), x(n/p)
DO I = 1, n/p
  global_I = I*p+my_pid
!!! sequential between processors
  recv B(global_I:I*p) from left
  x(I)=B(global_I)/L(global_I,I)
  DO J = global_I+1, I*p
     B(J) = B(J)-L(J,I)*x(I)
  END DO J
  send B(global_I+1:I*p) to right
!!! parallel between processors
  DO J = I*p+1, n
     B(J) = B(J)-L(J,I)*x(I)
  END DO J
!!! combine partial results in
!!! the dark gray area only
     global_sum(B(I*p+1:I*p+p),1:p)
 END DO I
```

Figure 2: Optimized Parallel Implementations of the Triangular Solver

3 Input Loop Forms

The inputs to the optimizing transformer are a class of data-parallel loops which can be formalized as *iterative spatial loop nests.*

Iterative Spatial Loops

A *iterative spatial loop nest* [2] consists of, zero or more levels of *iterative loops* (or *temporal loop*) followed by zero or more levels of *dependent spatial loops*, and then followed by one or more levels of *parallel spatial loops*, or simply *parallel loops*, as shown in Loop Nest I.

Once data are partitioned, spatial loop nests also need to be partitioned. A spatial loop is partitioned by splitting the loop into a processor loop and a local memory loop. The index in a processor loop gives the ID of the processor which computes the associated local memory loop.

Consider cyclic partitioning. A dependent spatial loop (DO_S loop) is split into a pair of local memory loop (DO_V for DO_S, or DOALL_V for DOALL_S) denoting the wrap-around layers in cyclic distribution and processor loop (DO_P or DOALL_P) denoting the active processors within a particular layer indexed by the DO_V/DOALL_V loop variable.

Loop Nest I (Iterative Spatial Loop Nest)

```
* * *iterative loops * **
DO_T ($K_1 = a_1, b_1, c_1$) {
...
DO_T ($K_l = a_l, b_l, c_l$) {
   * * *dependent spatial loops * **
   DO_S ($I_1 = x_1, y_1, z_1$) {
   ...
   DO_S ($I_m = x_m, y_m, z_m$) {
      * * *parallel loops * **
      DOALL_S ($I_{m+1} = x_{m+1}, y_{m+1}, z_{m+1}$) {
      ...
      DOALL_S ($I_{m+n} = x_{m+n}, y_{m+n}, z_{m+n}$) {
         $A(I_1, ..., I_{m+n}) = \tau[B(I_1 + c_1, ..., I_{m+n} + c_{m+n})]$     }}}}}}
```

Loop Nest 1 and Loop Nest 2 are the iterative spatial loop nest and partitioned loop nest for Program 1, respectively. The original arrays b, L, x are given new names $\hat{b}, \hat{L}, \hat{x}$ in the transformed loop. Let p be the number of processors and V_1 the size of local arrays. Index I is transformed to a pair of indices (I_1, I_2) denoting processors and local memory loops. The expression $I-1$ is transformed to $((g^{-1}(I_1, I_2) - 1) \bmod p, (g^{-1}(I_1, I_2) - 1)/p)$, which is further simplified to be $(I_1 - 1) \bmod p, I_2)$.

Loop Nest 1

```
DO_S ($I = 1, n$) {
   $x(I) = b(I, I-1)\ /\ L(I, I)$
   DOALL_S ($J = I+1, n$) {
      $b(J, I) = b(J, I-1) - L(J, I) * x(I)$  } }
```

Loop Nest 2

```
DO_V ($I_2 = 0, V_1 - 1$) {
   DO_P ($I_1 = 0, p-1$) {
      $I = I_2 * p + I_1$
      $\hat{x}(I_1, I_2) = \hat{b}(I, (I_1 - 1) \bmod p, I_2)$
      $\hat{L}(I, I_1, I_2)$
      DOALL_V ($J = I, n-1$) {
         $\hat{b}(J, I_1, I_2) = \hat{b}(J, (I_1 - 1) \bmod p, I_2)$
         $- \hat{L}(J, I_1, I_2) * \hat{x}(I_1, I_2)$            } } }
```

4 Transformation

Next we present the transformation that automates interleaving of reductions. We first describe the representation for data dependence, we then describe region analysis for extracting parallelism in reduction loops, Next, we present the transformation procedure.

4.1 Data Dependence Representation

Let d be the level of spatial loops in an iterative spatial loop nest. We use a vector $\delta = (a_1, \ldots, a_d)$ to represent a flow dependence in the loop nest. The vector element a_i contains the direction and distance of a flow dependence at the dimension corresponding to the ith loop. We divide the dependence distance into four classes:

- c (constant) : the dependence crosses a constant number c of iterations.
- v (variable) : the dependence distance is not a constant.
- r (reduction) : the dependence is artificial, caused by side-effecting the same data elements in reduction. In this work, we restrict reduction dependence distance to be 1.
- $[x : y]$ (interval) : the dependence distance is not a constant, but can be represented as an interval in either asscending order ($x < y$) or descending order ($x > y$). This kind of dependence distance may occur in a loop whose range is an affine function of an outer loop variable.

For example, the following loop contains two dependences.

```
DO_S (I = 1, m) {
    x(I) = a(I)
    DOALL_S (J = I + 1, n) {
        a(J) = a(J) - x(I)  }  }
δ1 = (0, -r), δ2 = ([-1 : n], -1)
```

4.2 Region Analysis

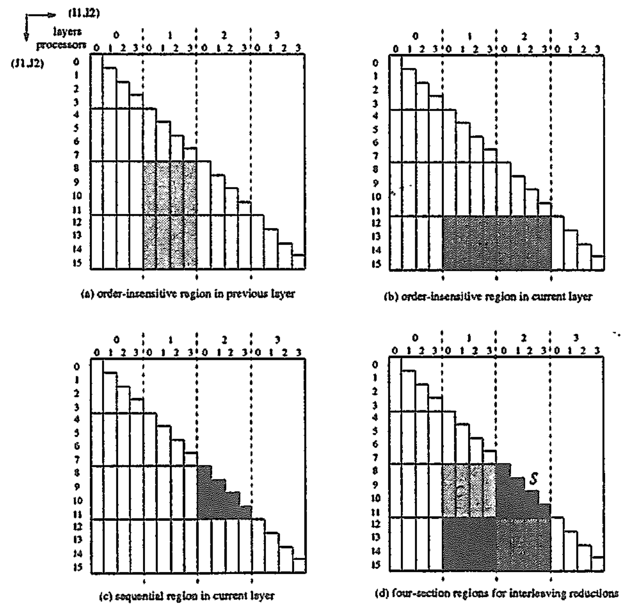

(I1,I2)
layers
processors
(J1,J2)
(a) order-insensitive region in previous layer
(b) order-insensitive region in current layer
(c) sequential region in current layer
S
(d) four-section regions for interleaving reductions

Figure 3: Region Analysis for Parallelizing Reduction in the Lower Triangular Solver

Figure 3 gives graphical intuition for partial parallelization of reductions, using the triangular solver as an example. We use the term "*order-insensitive region*" for the iterations in which the reduction terms can be summed up in any order. Such region can be identified straightforwardly using traditional data dependence analysis techniques. For example, in **Loop Nest 1** the order-insensitive region at iteration I can be denoted by the Cartesian product $\mathtt{interval}(I+1, n) \times \mathtt{interval}(1, I)$, corresponding to the DOALL_S(J) loop iterations and the DO_S(I) loop iterations respectively.

Figure 3 shows the order-insensitive regions between adjacent layers in the partitioned loop nest (Loop Nest 2). Figure 3(a) shows the order-insensitive region in layer $I_2 = 1$. Figure 3(b) shows the order-insensitive region in layer $I_2 = 2$. Figure 3(c) shows the *sequential region* where the reduction terms have to be summed up sequentially due to data dependence. Overlapping Figure 3(a),(b) and (c) results in Figure 3(d). Parallelism can be increased by distributing the reduction terms in region L (*parallel-update region*) over P processors and concurrently accumulating these reduction terms into those in region P. Region C (*global-reduction region*) contains the order-insensitive iterations whose results need be combined together in layer $I_2 = 1$ using a global reduction operation, so that the reduction result can be propagated to processor 0 before the computation for the *sequential region* S in layer $I_2 = 2$ can start. Detailed algorithm follows.

Algorithm region-analysis

input: dependence $[x : y]$, loop range $[Low : Upp]$, number of processors p.

output: sequential region S, global-reduction region C, parallel-update region L.

Case (1) $x < 0$, the sequential region $S = [Low : Low]$, the global-reduction region $C = [Low + 1 : Low + 1]$, and the parallel-update region $L = [Low + 2 : Upp]$.

Case (2) $x \geq 0$, $S = [Low : Low + \lceil\frac{(x+1)}{p}\rceil]$, $C = [Low + \lceil\frac{(x+1)}{p}\rceil + 1 : Low + \lceil\frac{(x+1)}{p}\rceil + 1]$, and $L = [Low + \lceil\frac{(x+1)}{p}\rceil + 1, Upp]$.

4.3 Transformation Procedure

We assume one level of DO_L and DO_P loops. For simplicity of presentation, in **Procedure interleaved-reduction** we assume loop ranges are in increasing order.

Procedure interleaved-reduction

Input: a partitioned loop nest and a set of dependence vectors δ_1,...,δ_d. Let $\delta_i(k)$ denote the kth element of δ_i.

1. If $\forall i \in [1..d]$ such that $\delta_i(k) = 0$ or $-r$, then this is a simple case where reduction can be interleaved straightforwardly: Replace the DO_P loop by a DOALL_P loop with the same range, replace reduction statements in the loop body by local memory updates, and insert a global reduction operation after the DOALL_P loop, then exit.

2. If $\exists i \in [1..d]$ such that $\delta_i(k) \neq 0$ or $-r$ and $\forall j \neq k, \delta_i(j) = 0$, or $-c$, or v, then exit, because no parallelism can be extracted due to data dependence.

3. An input loop nest that leads to this step suggests that data dependence, thought exist, is not parallel with the kth dimension, and therefore partial interleaving of reduction is possible. So, now there exist a j such that $\delta_i(j) = [x : y]$. Let J be the loop variable for the loop that carries this dependence, a be the coefficient of its loop range expression, p be the number of processors at the reduction dimension, and L be the loop variable for the local memory loop that carries the reduction. Strip-mine the J loop into two (indexed by $J_1 = L, ns - 1$ where ns is the number of strips, and $J_2 = 0, a * p - 1$) with strip size $a \times p$ so that all strips except the first one have the same size.

4. Call **region_analysis**($\delta_i(j)$,loop-range-of(J),p) to decide the sequential region, parallel-update region and global reduction region.

5. Split the range of the J loop into three corresponding to the sequential, global reduction, and parallel-update regions, and replicate the loop body. Move the global reduction loop and the parallel-update loop outside the sequential processor loop DO_P, enclose them with a new DOALL_P loop that has the same range as the DO_P loop. Replace the reduction statements enclosed by the new DOALL_P loop by local memory updates. Insert a global reduction operation at the end of the global reduction loop.

Example Strip-mining the DOALL_V loop in **Loop Nest 2** results in **Loop Nest 3**. The dependence vectors are $\delta_1 = (0, -r)$ and $\delta_2 = ([-1 : n], -1)$, therefore the sequential region includes loop range $J_1 = I_2, I_2$, the global reduction region includes loop range $J_1 = I_2 + 1, I_2 + 1$, and the parallel update region includes loop range $J_1 = I_2 + 2, ns - 1$. **Loop Nest 4** shows the result program after the transformations for interleaved reduction.

Loop Nest 3

```
DO_V ($I_2 = 0, v - 1$) {
    DO_P ($I_1 = 0, p - 1$) {
        $\hat{x}(I_1, I_2) = \hat{b}(I_2 * p + I_1, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)$ /
        $\hat{L}(I_2 * p + I_1, I_1, I_2)$
        DOALL_V ($J_1 = I_2, I_2$) {
            DOALL_V ($J_2 = I_1, p - 1$) {
                $\hat{b}(J_1 * p + J_2, I_1, I_2) =$
                $\hat{b}(J_1 * p + J_2, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)-$
                $\hat{L}(J_1 * p + J_2, I_1, I_2) * \hat{x}(I_1, I_2)$                } }
        DOALL_V ($J_1 = I_2 + 1, ns - 1$) {
            DOALL_V ($J_2 = 0, p - 1$) {
                $\hat{b}(J_1 * p + J_2, I_1, I_2) =$
                $\hat{b}(J_1 * p + J_2, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)-$
                $\hat{L}(J_1 * p + J_2, I_1, I_2) * \hat{x}(I_1, I_2)$                } }  } }
```

Loop Nest 4

```
DO_V ($I_2 = 0, v - 1$) {
    DO_P ($I_1 = 0, p - 1$) {
        $\hat{x}(I_1, I_2) = \hat{b}(I_2 * p + I_1, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)$ /
        $\hat{L}(I_2 * p + I_1, I_1, I_2)$
        * * *sequential region * **
        DOALL_V ($J_1 = I_2, I_2$) {
            DOALL_V ($J_2 = I_1, p - 1$) {
                $\hat{b}(J_1 * p + J_2, I_1, I_2) =$
                $\hat{b}(J_1 * p + J_2, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)-$
                $\hat{L}(J_1 * p + J_2, I_1, I_2) * \hat{x}(I_1, I_2)$                } }  }
    DOALL_P ($I_1 = 0, p - 1$) {
        combining region * **
        DOALL_V ($J_1 = I_2 + 1, I_2 + 1$) {
            DOALL_V ($J_2 = 0, p - 1$) {
                $\hat{b}((I_2 + 1) * p + J_2, I_1, I_2) =$
                $\hat{b}((I_2 + 1) * p + J_2, (I_1 - 1)$ MOD $p, I_2 + (I_1 - 1)$ DIV $p)-$
                $\hat{L}((I_2 + 1) * p + J_2, I_1, I_2) * \hat{x}(I_1, I_2)$                  }
            global sum operation
            DOALL_V ($J_2 = 0, p - 1$) {
                $\hat{b}((I_2 + 1) * p + J_2, I_1, I_2) =$ sum$(\hat{b}((I_2 + 1) * p + J_2, 0 : p - 1, I_2))$  } }
        * * *local update region * **
        DOALL_V ($J_1 = I_2 + 2, ns - 1$) {
            DOALL_V ($J_2 = 0, p - 1$) {
                $\hat{b}(J_1 * p + J_2, I_1, I_2) = \hat{b}(J_1 * p + J_2, I_1, I_2 - 1)$
                $- \hat{L}(J_1 * p + J_2, I_1, I_2) * \hat{x}(I_1, I_2)$                } }          }
```

5 Experimental Study

We evaluate the effectiveness of this optimization by conducting experiments on the Connection Machines CM-5 located in the University of Minnesota and the nCUBE2 located in the Institute of Information Science, Academia Sinica. Two benchmark programs were implemented: a lower triangular solver, and a LU solver.

Table 1 and Table 2 show the effectiveness and scaled speedup of interleaved reduction optimization on the lower triangular solver. The sequential times are given as a basis for comparison. The unoptimized version implemented the two nested loops given in **Program 1** using cyclic distribution.

The speedup of the optimized version against the unoptimized version increase as the problem size increases. When per-processor problem size is fixed, the speedup factor decreases a little bit when number of processors increases. This is because the extra overhead in global reduction operation increases with the number of processors.

Table 3 compares the performance of the LU solver on 32-node CM-5. The LU solver solves a lower triangular system and an upper triangular system. The CM-Fortran version implemented the traingular solvers using nested loops as shown in **Program 1** and was compiled by the CM-Fortran compiler 2.1.2 with the -O -vu option. The CMSSL LU solver is a hand-crafted, micro-coded library routine by Thinking Machines. The optimized version (with interleaved reductions transformation) outperformed the CM-Fortran compiler by two orders of magnitude, and achieved more than 50% performance of the CMSSL routine.

Problem size	seq	unopt	opt	$\frac{seq}{opt}$	$\frac{unopt}{opt}$
128×128	0.01	0.06	0.05	0.25	1.20
256×256	0.04	0.15	0.08	0.51	1.93
512×512	0.16	0.41	0.12	1.32	3.38
$1k \times 1k$	0.64	1.44	0.24	2.68	6.05
$2k \times 2k$	2.56	5.69	0.47	5.39	11.98
$4k \times 4k$	10.26	21.90	0.93	11.01	23.49

Table 1: Execution time in seconds for a parallel triangular solver on 32-processor CM-5, double precision

nproc	seq	unopt	opt	$\frac{seq}{opt}$	$\frac{unopt}{opt}$
32	10.26	21.90	0.93	11.03	23.49
64	20.76	44.77	1.99	10.43	22.41
128	42.98	91.78	4.40	9.77	20.84
256	85.16	188.33	9.87	8.63	19.06
512	173.32	391.23	22.15	7.82	17.66

Table 2: Execution time in seconds for a parallel triangular solver on CM-5, with fixed per-processor problem size ($512k$), double precision

Problem size	CMF	opt	CMSSL	$\frac{CMF}{opt}$	$\frac{opt}{CMSSL}$
128×128	1.48	0.11	0.05	13.45	2.20
256×256	4.99	0.17	0.08	29.35	2.12
512×512	20.04	0.25	0.14	143.14	1.78
$1k \times 1k$	82.57	0.48	0.25	172.02	1.92
$2k \times 2k$	334.27	0.93	0.48	359.43	1.93
$4k \times 4k$	-	1.96	0.99	-	1.97

Table 3: Execution time in seconds for LU solver on 32-processor CM-5, double precision

Table 4 shows the execution time of the lower triangular solver on 32-node nCUBE2. Consistent with the results on the CM-5, the speedup factors on the nCUBE2 also increases with the problem size.

The results of the lower triangular solver and the LU solver demonstrate that interleaved reduction can improve program performance significantly, especially for large problem sizes. On machines with fast hardware support for global reductions, such as the CM-5, this optimization also scales up reasonably, though not perfectly, to large machine sizes.

Problem size	seq	unopt	opt	$\frac{seq}{opt}$	$\frac{unopt}{opt}$
256×256	0.08	0.31	0.09	0.89	3.44
512×512	0.34	1.10	0.17	2.00	6.47
$1k \times 1k$	1.35	4.13	0.35	3.86	11.8
$2k \times 2k$	5.60	16.10	0.83	6.75	19.40

Table 4: Execution time in seconds for a parallel triangular solver on 32-node nCUBE2, double precision

6 Conclusion

In this paper, we have presented an interleaving transformation for optimizing reductions on massively parallel machines. The optimization exploits relatively fine-grain parallelism that is suitable for most massively parallel computation platforms. Our experiences with an LU solver on the Connection Machine CM-5 give us confidence that such automatic transformation can achieve significant performance improvement over straightforward implementations.

References

[1] P. Bose. Interactive program improvement via eave. In *Proceedings of the International Conference on Supercomputing*, 1988.

[2] Marina Chen and Yu Hu. Optimizations for Compiling Iterative Spatial Loops to Massively Parallel Machines. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, 1992.

[3] 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*, 1994.

[4] B. Leasure. The parafrase project's fortran analyzer. Technical Report 85-504, Dept. Computer Science, University of Illinois at Urbana-Champaign, 1985.

[5] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *Proceedings of Principles of Programming Languages*, 1990.

[6] X. Roden and P. Feautrier. Detection of recurrences in sequential programs with loops. In *Lecture Notes in Computer Science, vol. 694*, 1993.

[7] Chau-Wen Tseng. *An Optimizing Fortran D Compilers for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, 1993.