An Interleaving Transformation for Parallelizing Reductions for Distributed-Memory Parallel Machines

JAN-JAN WU
 Institute of Information Science, Academia Sinica, Taipei Taiwan 11529, Republic of China

Received January 1999; accepted March 1999

Abstract. Reduction operations frequently appear in algorithms. Due to their mathematical invariance properties (assuming that round-off errors can be tolerated), it is reasonable to ignore ordering constraints on the computation of reductions in order to take advantage of the computing power of parallel machines.

One obvious and widely-used compilation approach for reductions is syntactic pattern recognition. Either the source language includes explicit reduction operators, or certain specific loops are recognized as equivalent to known reductions. Once such patterns are recognized, hand optimized code for the reductions are incorporated in the target program. The advantage of this approach is simplicity. However, it imposes restrictions on the reduction loops—no data dependence other than that caused by the reduction operation itself is allowed in the reduction loops.

In this paper, we present a parallelizing technique, interleaving transformation, for distributed-memory parallel machines. This optimization exploits parallelism embodied in reduction loops through combination of data dependence analysis and region analysis. Data dependence analysis identifies the loop structures and the conditions that can trigger this optimization. Region analysis divides the iteration domain into a sequential region and an order-insensitive region. Parallelism is achieved by distributing the iterations in the order-insensitive region among multiple processors. We use a triangular solver as an example to illustrate the optimization. Experimental results on various distributed-memory parallel machines, including the Connection Machines CM-5, the nCUBE, the IBM SP-2, and a network of Sun Workstations are reported.

Keywords: reduction, program transformation, parallel processing, distributed-memory, parallelizing compilation, triangular systems solver

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

\[
\text{DO } I = 1, n \\
\quad \text{sum} = \text{sum} + X(I) \\
\text{END DO}
\]
presents a problem to compilers, because dependence 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, computing the \( \text{SUM} \) of \( n \) values can be parallelized using \( p \) processors as described below. First, divide the \( n \) values into \( p \) chunks and assign one chunk to every processor. Then let every processor compute locally the sum of the values assigned to it, independent of other processors. Finally, combine the partial results produced by these processors. Figure 1 illustrates some parallelization strategies for computing the reduction \( \text{SUM}(X) = \sum_{i=1}^{n} X(i) \) on two processors, \( P0 \) and \( P1 \). In Figure 1(b) and Figure 1(c), array \( X \) is distributed to processors \( P_0 \) and \( P_1 \) in a wrap-around fashion, which we will refer to as interleaving.

Parallelization of reductions require language constructs or compiler techniques that can help break data dependence in reductions. One obvious compilation approach is pattern recognition. Either the source language includes explicit reduction operators (e.g., the \( \text{SUM} \) operator in High Performance Fortran [12]), or certain specific loops are recognized as equivalent to known reductions (e.g., most parallelizing compilers recognize the loop for \( \text{SUM} \) reduction described above). Once such patterns are recognized, the compiler replaces the patterns with hand optimized code for the reductions.

Previous work on pattern recognition for reductions had been reported in the Parafrase system [10], the Eave [3] system, the Vienna Fortran system [23], the Fortran D system [21], the Fortran 90D/HPF system [4], the SUIF compilation system [8], the KAP Fortran optimizer [1], and the MIPSPro Power Fortran compiler [2]. Redon and Feautrier proposed an algebraic specification method for

\[
\begin{align*}
\text{PO} &\quad \text{P1} \\
\text{Compute partial sums in parallel} &\quad \text{sum0} = X(1)+X(2)+X(3)+X(4) &\quad \text{sum1} = X(5)+X(6)+X(7)+X(8) \\
\text{Combine partial sums} &\quad \text{sum} = \text{sum0} + \text{sum1}
\end{align*}
\]

(a) parallelization by partitioning \( X(1:8) \) into two contiguous blocks (i.e. using block distribution)

\[
\begin{align*}
\text{PO} &\quad \text{P1} \\
\text{Compute partial sums in parallel} &\quad \text{sum0} = X(1)+X(3)+X(5)+X(7) &\quad \text{sum1} = X(2)+X(4)+X(6)+X(8) \\
\text{Combine partial sums} &\quad \text{sum} = \text{sum0} + \text{sum1}
\end{align*}
\]

(b) parallelization by interleaving \( X(1:8) \) onto two processors (i.e. using cyclic distribution)

\[
\begin{align*}
\text{PO} &\quad \text{P1} \\
\text{Compute partial sums in parallel} &\quad \text{sum0} = X(1)+X(2)+X(5)+X(6) &\quad \text{sum1} = X(3)+X(4)+X(7)+X(8) \\
\text{Combine partial sums} &\quad \text{sum} = \text{sum0} + \text{sum1}
\end{align*}
\]

(c) parallelization by interleaving \( X(1:8) \) onto two processors with block size=2 (i.e. using generic block-cyclic distribution)

*Figure 1.* Some parallelization strategies for the \( \text{SUM} \) reduction.