Compile-Time Loop Splitting for Distributed Memory Multiprocessors

Donald O. Tanguay, Jr.*

MIT Laboratory for Computer Science
545 Technology Square, Room NE43-633
Cambridge, Massachusetts 02139

1 Abstract

In a distributed memory multiprocessor, a program's task is partitioned among the processors to exploit parallelism, and the data are partitioned to increase referential locality. Though the purpose of partitioning is to shorten the execution time of an algorithm, each data reference can become a complex expression based upon the data partitions. As an attempt to minimize the computation needed for array references, loop splitting can further divide a partitioned loop into segments that allow the code hoisting and strength reduction optimizations. This thesis introduces two methods of loop splitting, *rational* and *interval*. While rational splitting divides the loop into equal-length GCD segments, interval splitting specifies segments as an explicit list of intervals. These two methods have been implemented and studied. Under our execution model, the loop in the algorithms analyzed executes an average of 2 to 3 times faster after loop splitting.

2 Introduction

Distributed memory multiprocessors have several advantages over shared memory machines, such as scalability and nonuniform access time. However, the distributed memory of the machine introduces the problems of finding and exploiting locality, distributing task and data of a program, and addressing data.

The addressing complications are manifested in the involved expressions associated with array referencing. To reference a particular array cell, the expression must calculate not only the memory location of the cell but also which processor's memory contains the cell. These two address specifications are functions of the array distribution and the network configuration.

Methods of array referencing on distributed memory multiprocessors vary; the method described here is used by the Alewife effort ([1]) at MIT. Array referencing in Alewife is implemented in software, and the general expression to access an array cell is

$$\text{aref (aref } (array,\ pid),\ offset)\ ,$$

where aref is the array reference procedure, $array$ is the name of the array, pid is the unique ID of the processor whose memory contains the cell, and $offset$ is the offset into that memory. A deeper description of the array references, as well as the other subjects in this paper, can be found in [2].

*E-mail address: donald@hing.lcs.mit.edu. This research was supervised by Professor Anant Agarwal and Dr. David Kranz. It was funded in part by NSF grant # MIP-9012773 and in part by DARPA contract # N00014-87-K-0825.

Source Code	Object Code
for(*i*,LL,UL,1) {temp = A[*i*];}	**for**(*i*,LL,UL,1) {temp = aref (aref (A, *i*/i_spread), *i*%i_spread);}

Table 1: The source code and object code of the general 1-D array reference.

Table 1 shows the source and object code for a general 1-D array reference A[i]. The constants i-proc, i-spread, and j-spread are parameters of the data partitioning of the array and of the processor network. The variables I and J are the induction variables of the loops containing the reference expressions.

3 The Problem

In rectangular task and data partitioning, possibilities exist for simplifying the array reference expressions in loops. When the data in a loop is located on only one processor, the processor ID is loop invariant (i.e., has a constant value for the duration of the loop), and the offset is quasi-invariant (can be computed with counters rather than remainder expressions). To avoid recomputing these invariant parts of the address on every access, a compiler can perform code transformations that simplify the expressions. Once performed, these transformations allow even further code improvements by an optimizing compiler. The result is a less computationally expensive loop body that reduces the execution time of the loop.

Unfortunately, because the memory is distributed, the data needed by a loop is very often located on more than one processing element so that no loop invariants exist for optimization. However, because arrays are often both accessed and distributed in segments of contiguous array cells, *intervals* of a loop access data from a single processor and have their own invariants. By dividing the loop into these intervals, the code transformations can still be performed, albeit on a smaller scale.

4 Loop Splitting

A compiler can isolate these intervals by performing a loop transformation called *loop splitting* that divides a loop into subloops. The loop invariants are exposed to optimization so that not only are the array reference operations fewer in number but they are also no longer performed on every iteration.

The code optimizations allowed by loop splitting are code hoisting and strength reduction. The *code hoisting* compiler optimization

Optimized Code
for(I_step,LL,UL,i_spread){
I_div = I_step/i_spread;
I_rem = I_step%i_spread;
for(*i*,I_step,I_step+i_spread,1)
{temp = aref (aref (A, I_div), I_rem);
I_rem++;}}

Table 2: The optimized general 1-D array reference.

for loops, also known as *factoring loop-invariants*, avoids redundant computations by pulling an expression out of a loop body to precede the loop code. In the array reference expressions, computation of the processor ID is removed from the loop body. The *strength reduction* optimization replaces an expensive (with regards to time) operation with a less expensive one. In the array reference expressions, a compiler can reduce in strength the complex modulo function, which requires repeated divisions until a remainder is found, into a faster variable increment.

Table 2 shows the previous object code after loop splitting. The loop splitting interval is based directly on the partitioning of the array A. In general, however, the interval is a more complex expression based on multiple data partitions. In such cases, it is harder for a compiler to determine the iteration intervals for which the expressions have invariants.

Although loop splitting improves array references, the extent of the improvement is unclear. We attempted to clarify this issue by experimenting with two loop splitting methods on several benchmarks.

5 Implemented Methods

We have introduced two methods of loop splitting that improve all array references of a loop – rational splitting and interval splitting. While rational splitting creates subloops of length equal to the greatest common divisor of the data spreads, interval splitting explicitly defines the subloops in order to remove all redundant computations.

Rational splitting divides each loop in the original loop nest into subloops with the same number of iterations. To group the invariants and minimize the amount of redundant computation, the subloop size is the greatest common divisor of the data partition dimensions. Thus, a single number (the gcd) specifies how the loop is split; this minimal information keeps code size small. Unfortunately, it is not uncommon that the gcd of the data spreads is one. In such a case, no references are optimized, and the resulting code is slower than the original due to the extra loop overhead on every iteration.

Interval splitting is a different approach to the problem of multiple data partitions. Where rational splitting uses a single number to divide a loop into subloops, interval splitting uses a *list* of numbers, explicitly specifying the subloops. With this list, the compiler has already decided exactly how to divide the loop for each processor so that *all* redundant computations are avoided. Unfortunately, because each processor can have a unique list of intervals, as the number of processors increases the code size also increases. With a 256 processor network, for example, the case statement assigning the interval lists contains 256 lists. On a thousand-node network, very simple code can become alarmingly large.

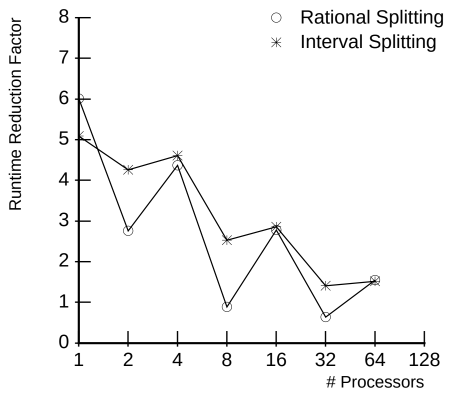

Figure 1: Performance improvement of a transposition algorithm for 100x50-element matrices.

6 Results

Implementations of the two methods were created as source-to-source transformations and added to the Alewife compiler to analyze the improvement of loop execution time. Ignoring the effects of caching and communication, the results on four different benchmarks in two compiler settings were gathered. The programs used to analyze loop splitting effects were chosen to be representative of the various routines found in scientific code. These programs are vector addition, matrix addition, matrix multiplication, and matrix transposition. Improvement was computed as the ratio of the original loop's execution time to that of the loop after loop splitting. The number of processors was varied in the experiments because it determined the task and data partitions, in turn determining the data spreads that affect the loop splitting intervals.

For brevity, Figure 1 shows the performance improvement of only one experiment, a matrix transposition algorithm. The average improvement, as for the other algorithms, is a 2- to 3-fold reduction in execution time. Figure 1 also shows clearly the effects of "incongruent" data partitions, where improvement decreases due to similar-but-not-identical data partitions with low gcd's. Explicitly specifying all intervals, interval splitting is more resistant to this jagging effect.

Results show that loop splitting is an effective technique to lower the computation cost of array references on a distributed memory multiprocessor. In most cases, it is easy to improve the execution time of a loop to 2 to 3 times faster.

The rational splitting method has code size independent of the number of processors but has good probability of reintroducing much of the array reference computations. In some cases, where data partitions were similar but not identical, rational splitting performs worse than the original code. Interval splitting, on the other hand, minimizes the number of computations but has the potential of large code size with a high number of processors. However, incongruent data partitions still greatly diminish its effectiveness.

References

[1] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multip rocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiproce ssors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publicati on, and appears as MIT/LCS Memo TM-454, 1991.

[2] Donald O. Tanguay Jr. Compile-time loop splitting for distributed memory multiprocessors. MIT Bachelor's Thesis, May 1993.