Compiling SIMD Programs for MIMD Architectures

Abstract

Programming multiple-CPU computers is widely held to be harder than programming sequential computers, but much of the difficulty can be traced to the MIMD programming languages used. SIMD languages provide programmers with a more understandable model of parallel computation. In this paper we summarize the advantages of data parallel languages, a subclass of SIMD languages, and describe how programs written in a data parallel language can be compiled into loosely-synchronous MIMD programs suitable for efficient execution on multicomputers.

1 Introduction

Flynn’s [2] taxonomy of computer architectures is the basis for a variety of programmer models of parallel computation. To a programmer, an SIMD (single instruction stream, multiple data stream) computer can be viewed as a single CPU directing the activities of a number of arithmetic processing units, each capable of fetching and manipulating its own local data. In any time unit a single operation is in the same state of execution on multiple processing units, each manipulating different data. Hence this programming model is called synchronous.

In contrast, an MIMD (multiple instruction stream, multiple data stream) computer allows the concurrent execution of multiple instruction streams, each manipulating its own data. It is possible for every processor in an MIMD computer to execute a unique program, but it is far more common for every processor to execute the same program. This is known as SPMD (single program, multiple data stream) programming. Although processors may coordinate with each other at synchronization points, the MIMD and SPMD programming models are called asynchronous, because between the synchronization points every processor executes instructions at its own pace.

A number of SIMD programming languages have been proposed, including Actus [6], C* [10], Parallel Pascal [8], and Vector C [5]. We believe an SIMD programming language should have the following features: a global name space, which obviates the need for explicit message passing between processing elements; parallel objects, rather than merely parallel scalars; and the ability to make the number of processing elements a function of the problem size, rather than a function of the target machine. We use the term data parallel to refer to SIMD languages with all these properties [4]. Of the languages we have listed, only C* fits our definition as a data parallel language.

Earlier work on SIMD programming languages has concentrated on developing compilers for processor arrays or pipelined vector processors [5,6]. Our thesis is that compilers can translate SIMD programs into programs that run efficiently on a wide variety of architectures. In this paper we describe how data parallel programs can be translated into SPMD programs that execute efficiently on multicomputers.

Our work is closely related to recent efforts that translate to multicomputers sequential programs that are augmented with data distribution information [1,9]. We feel that if the language is to contain data mapping constructs, then the language should be explicitly parallel. An SIMD modeling program maintains the benefits of using a sequential language (deterministic results, no race conditions, ability to debug on a von Neumann architecture) while allowing the natural expression of inherently parallel operations (such as data reductions). In addition this paper will demonstrate that an SIMD program may not have a strictly synchronous execution. Just as sequential programs can be mapped to asynchronous parallel programs with the same behavior, we map SIMD programs to equivalent asynchronous implementations.

Earlier we presented the design of a C*-to-C compiler based upon a general control flow model [7]. This paper describes a new compiler design based upon a less general control flow model, i.e., one that does not support the goto statement. Experimental results show that the new compiler generates programs with significantly higher efficiency, programs that often rival and sometimes match the speed of hand-coded C programs on a hypercube multicomputer.

Section 2 summarizes the strengths of the data parallel approach to problem solving on parallel computers. Section 3 presents a brief overview of C*, the data parallel language used in our examples. In Section 4 we describe how data parallel programs can be translated into efficient SPMD programs suitable for execution on a hypercube multicomputer. Experimental results appear in Section 5.

2 Strengths of the Data Parallel Model

The data parallel model of computation has a number of inherent strengths. Some of these strengths are common to all SIMD languages, while others are dependent upon the data parallel model’s particular attributes.

Several advantages relate to the model’s simple control flow. It is easier to determine the state of the system, since all pro-
In order to express the parallel program segments in this paper, we adopt the conventions of the data parallel programming language C*, developed by Thinking Machines Corporation [10]. C* is a superset of the sequential programming language C. In this section we briefly describe a few important features of C*.

For further details, see the paper by Rose and Steele.

All data in C* are divided into two kinds, scalar and parallel, referred to by the keywords mono and poly, respectively. C* allows the programmer to express algorithms as if there were an unbounded number of processors onto which the data can be mapped. Once every piece of parallel data has been mapped to its own processing element, several simple program constructs allow parallel operations to be expressed. The most important of these constructs is an extension of the class type in C++ [11]. A class is an implementation of an abstract data type. Instances of variables of a particular class type are manipulated with that class's member functions. In C* member functions operate on a number of instances of a class in parallel. This "parallel class type" is called a domain.

In C* variables of a domain type are mapped to separate processing elements, and all instances of a domain type may be acted upon in parallel by using that domain's member functions and the selection statement (which is illustrated below). Within parallel code each sequential program statement is performed in parallel for all instances of the specified domain.

The following code segment computes the element-wise maximum of two arrays.

```c
domain vector { float a, b, max; } x[100];
< Intervening code >
[domain vector] { if (a > b) max = a;
else max = b; }
```

The domain type vector defines a domain containing two floating point values named a and b. By declaring x to be a 100-element array of vector, 100 instances of the variable pair are created, one pair for each processing element. The selection statement [domain vector] activates every processing element whose instance has domain type vector; i.e., every element of x. Every active processing element executes the statements contained within the selection statement. In this case every processing element evaluates the expression a > b. The universal program counter enters the then clause, and those processing elements for which the expression is true perform the assignment statement `max = a'. Next the universal program counter enters the else clause, and those processing elements for which the expression is false perform the assignment statement `max = b'.

C* programs have a single name space, and any expression can contain a reference to any variable in any domain. For example, consider the following code segment, in which every active processing element sets its own value of temp to be the average

<table>
<thead>
<tr>
<th>Application</th>
<th>C</th>
<th>C*</th>
<th>NCUBE C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiplication</td>
<td>16</td>
<td>16</td>
<td>85</td>
</tr>
<tr>
<td>Prime sieve</td>
<td>23</td>
<td>25</td>
<td>78</td>
</tr>
<tr>
<td>Mandelbrot set</td>
<td>31</td>
<td>31</td>
<td>89</td>
</tr>
<tr>
<td>Amortization table</td>
<td>31</td>
<td>34</td>
<td>131</td>
</tr>
<tr>
<td>Moore's shortest path</td>
<td>48</td>
<td>53</td>
<td>294</td>
</tr>
<tr>
<td>Bitonic merge sort</td>
<td>50</td>
<td>51</td>
<td>149</td>
</tr>
<tr>
<td>Dijkstra's shortest path</td>
<td>64</td>
<td>71</td>
<td>176</td>
</tr>
<tr>
<td>Path consistency</td>
<td>75</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>Character thinning</td>
<td>86</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>98</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>Fluid flow</td>
<td>182</td>
<td>182</td>
<td>345</td>
</tr>
<tr>
<td>Ray tracing</td>
<td>503</td>
<td>517</td>
<td>661</td>
</tr>
</tbody>
</table>

Table 1: Comparison of lines of code needed to program various applications in the sequential language C, the data parallel language C*, and the MIMD parallel C language on the NCUBE. NCUBE C figure reflects node program only.
of the temp values of its predecessor and successor processing elements:

```c
#define N 100
domain rod { float temp; } x[N];
< Intervening code >
[domain rod]. { 
  int index = this-x; /* this same as in C++ */
  < Intervening code >
  if ((index > 0) && (index < N-1))
    temp = (x[index-1].temp + x[index+1].temp)/2;
}
```

Each processing element's value of index gives its unique position in the domain, a value in the range 0..N-1. All active processing elements evaluate the right hand side of the assignment statement together, then they all perform the assignment of values together. Hence an old value cannot be overwritten before an adjacent processing element has had the opportunity to read it.

4 The Translation of SIMD Programs Into SPMD Programs

We shall focus on three important issues. The first issue is the translation of control structures. Because synchronization is relatively expensive on contemporary multicomputers, the compiler must minimize the number of synchronizations required as the parallel computer works its way through the various control structures of the data parallel program. The second issue is the emulation of processing elements. Data parallel programs often assume a very large number of processing elements. If these programs are to run efficiently on a multicomputer with far fewer physical processors, then there must be an efficient mechanism for emulating processing elements on physical processors. The third issue is message passing optimization. Because message-passing overhead is often significant on contemporary multicomputers, the compiler must take every opportunity to concatenate multiple short messages into a single, longer message.

4.1 Minimizing Processor Synchronizations

Any efficient implementation of an algorithm on a multicomputer must simultaneously balance the work among the physical processors and minimize the number of processor interactions. In our case, the workload is balanced through an appropriate mapping of processing elements (e.g., data structures) to physical processors. Automatic mapping of data to processors to balance the workload and minimize interprocessor communications is beyond the capabilities of contemporary compilers [12], and we assume the programmer has the means to assist the compiler. Automatic methods of minimizing processor interactions have been studied with more success, and in the following paragraphs we explain how a SIMD-to-SPMD compiler can perform this task.

Assuming that we spread processing elements around the processors of the multicomputer, and that the sequential code is executed by one additional special-purpose processing element, the potential communication requirements of an SIMD program can be traced to those points at which processing elements read or write values to or from each other. Recall that in the C* language expressions can refer to values stored in arbitrary processing elements. All variable declarations state, implicitly or explicitly, which processing element holds the variable being declared. Therefore, the location of potential communication points can be reduced to a type checking problem, where the declarations for identifiers appearing in an expression are accessed to determine the meaning of the expression.

We approach the synchronization issue by asking the following question: how much can we loosen the synchronization requirements of the language without affecting the behavior of programs? If a set of processing elements were executing the same block of code and were only accessing their own local variables, it would not matter if they synchronized after every statement, after every other statement, or only at the end of the block. However, if halfway through the block each processing element reads a value from its neighbor, the processors must be synchronized prior to accessing their neighbor's memory, in order to guarantee that the value retrieved by each processor is the same as it would be under a fully synchronous implementation. Hence the synchronization points are the points at which we identify message passing is potentially needed. Synchronization is incorporated into the message passing primitives.

Parallel looping constructs may require additional synchronization. If the loop body does not require any message passing, then no synchronization is necessary for the loop. Since the processing elements executing the loop do not interact, it does not matter if they are executing different iterations of the loop concurrently. Active processing elements only need to synchronize just before they interact.

If the parallel loop body does require message passing, the processors must be synchronized every iteration prior to the message passing step. Because our message passing primitives require the active participation of all physical processors, we must guarantee that if any processing elements are still executing the loop, all physical processors must be executing at least the message passing primitive required by the loop. In other words, no processing element can execute the statement after the loop until all processing elements have exited the loop. This constraint forces all processing elements to communicate each iteration in order to compute a global logical OR of the processing elements' Boolean loop control values. A processing element can exit the loop only when the global logical OR is false. Since this global OR operation is only required when there is message passing occurring inside the loop, we bundle the global OR operation in with the required message, allowing us to say that the points at which messages are being sent are exactly the points at which the processors perform a barrier synchronization.

Of course, a processing element does not actually execute the body of the loop after its local loop control value has gone
The C* construct:

```
while (condition) {
    statement_list1;
    communication;
    statement_list2;
}
```

is translated into the following C code:
```
temp = TRUE;
do {
    if (temp) {
        temp = condition;
    }
    if (temp) {
        statement_list1;
    }
    communication;
gtemp = global_or(temp)
    if (temp) {
        statement_lists;
    }
} while (gtemp);
```

Figure 1: Translation of C* while loop.

to false. Rather, the physical processor on which it resides participates in the message passing and the global OR operation. This means that our C* compiler must rewrite the control structure of input programs. Figure 1 illustrates how while loops are rewritten.

The requirement that all physical processors must actively participate in any message passing operation forces our compiler to rewrite all control statements that have inner statements requiring message passing. We must rewrite the statement in order to bring the message passing operations to the surface of the control structure. Figure 2 illustrates how an if statement is handled.

Communication steps buried inside nested control structures are pulled out of each enclosing structure until they reach the outermost level. Figure 3 shows a nested if statement.

The technique just described will not handle arbitrary control flow graphs. For this reason we have not implemented the goto statement. We can, however, handle the break and continue statements.

We want to emphasize that processor synchronizations occur only when processing elements interact. No additional synchronizations are needed to support control flow. This is the primary reason why our new compiler generates more efficient code than our first compiler [7].
4.2 Efficiently Emulating Processing Elements

Once the message-passing routines have been brought to the outermost level of the program, emulation of processing elements is straightforward. The compiler puts for loops around the blocks of code that have been delimited by message passing/synchronization steps. Since within the delimited blocks there is no message passing, there is no interaction between processing elements. It makes no difference semantically in which order the processing elements located on a particular physical processor execute.

4.3 Optimizing Message-Passing

A primary goal of the compiler is to minimize the number of messages passed, because message initiation is a relatively expensive operation on a multicomputer. This section discusses a set of message-passing optimizations. We expect to add to this list as we get further experience with C* programs.

One class of messages is eliminated by keeping copies of the sequential code and data on each physical processor. Every physical processor executes the sequential code. Note that this adds nothing to the execution time of the program. If a single physical processor executed the code while the other physical processors sat idle, the execution time would be the same. Because every physical processor has copies of the sequential variables, it can access sequential data by doing a local memory fetch. In other words, assigning the value of a mono variable to a poly variable can be done without any message passing.

By putting a copy of the sequential code and data on each physical processor, we are assuming that mono values are retrieved more often than they are assigned. With our design the retrieval of a mono is free—every physical processor has a local copy of the mono that can be accessed by a processing element without message passing. However, when a processing element stores to a mono variable, the value may have to be broadcast to update all physical processors' copy of the mono. If an analysis of the control flow indicates that all processing elements are active for the store to a mono, and the value being stored is known at compile time to be the same on all processing elements, then the compiler can omit the subsequent broadcast.

The tracking of mono values through a simple data flow analysis can be used to "sequentialize" loop constructs to avoid possible loop synchronizations. For instance, if the loop control variable is initialized to a mono value, the loop termination expression is known to evaluate to the same value on all processing elements, and the loop increment step is the same for all virtual processors, then the compiler knows in advance that all virtual processors will execute the loop the same number of times.

No global OR of the loop termination expression value need be done. In addition, no virtual processor emulation of the control portion of the loop structure is required. The for loop that is introduced to emulate virtual processors can be moved inside of the C* loop.

This type of analysis should also be applied to expressions that require message passing. If the compiler determines that all processing elements are requesting the same piece of "non-local" data, then the data should be requested only once by a physical processor and shared among the resident processing elements.

Message-passing expressions should be examined by a common subexpression detection phase of the compiler. Since C* programmers refer to "non-local" values by writing an expression, equivalent expressions should cause the compiler to analyze the intervening code to see if it could affect the value of the expression. If not, then the distant value should only be retrieved once with the expensive message-passing step, and a local copy of the value should be kept for the second reference.

Another optimization is performed by viewing message passing as a possible vector operation. That is, if a block of data (such as a row of a matrix) must be sent to another processor, the whole block can be sent in one message, rather than as a series of shorter messages. The amount of data transferred is essentially identical, but the message-passing overhead is drastically reduced. Since this is a vectorization operation, it can be performed using well-known techniques designed to be used with sequential machines that have vector instructions.

A similar optimization can be performed upon the code generated to support processing elements. Instead of each active processing element sending an individual message, the message bodies of all active processing elements can be gathered together and shipped at once. The motivation once again is to reduce the number of communication steps.

4.4 Summary

To summarize, the compiler must first locate the points at which message passing is required. These points are identical to the synchronization points. Therefore, our message passing primitives also synchronize the processors. Second, the compiler must transform the control structure of the input program to bring message-passing primitives to the outermost level. In order to allow a single physical processor to emulate a number of processing elements, the compiler must insert for loops around the blocks of code that are delimited by the calls to the message passing primitives. Finally, data flow analysis can be used to eliminate some calls on message-passing routines and to combine multiple shorter messages into single, longer messages whenever possible.

5 Experimental Results

To demonstrate the efficiency that can be achieved by SIMD programs executing on a hypercube multicomputer, we present in Figure 4 the speedup achieved by three hand-compiled C* programs. The three programs are: Mandelbrot set calculation, matrix multiplication, and Gaussian elimination. The Mandelbrot program computes the colors of a $128 \times 128$ image representing a square of the complex plane centered around the origin and having sides of length 4. The matrix multiplication program multiplies two $128 \times 128$ integer matrices. The Gaussian elimination program reduces a dense system of 256 linear equations with 256 unknowns.
Figure 4: Speedup achieved by three hand-compiled C* programs executing on an NCUBE/seven multicomputer.

The first two programs—Mandelbrot set calculation and matrix multiplication—have a high degree of parallelism and a simple control structure. It is relatively straightforward for the C* compiler to generate code comparable in performance to hand-written C code. The third C* program, Gaussian elimination with partial pivoting, has a good measure of sequentiality to it, and processing elements exchange data frequently. In order to produce efficient target code, a compiler must be able to perform numerous message-passing optimizations. After performing the code optimizations described in the previous section, the resulting Gaussian elimination program is equal in performance to the best known hand-coded C program on the NCUBE.

6 Summary

We have discussed how a compiler can translate data parallel SIMD programs into SPMD-style MIMD programs and have presented evidence that high efficiency is an attainable goal. We are in the process of implementing a C*-to-C compiler for the NCUBE series of hypercube multicomputers. At this time the compiler translates C* programs into SPMD-style C programs, but it does not perform all the message passing optimizations necessary to produce highly efficient C code. Once the optimizing compiler has been completed, we will use it to help us determine which problems seem most suitable for solution via data parallel algorithms.

We do not see data parallel programming as the only solution to the problem of programming MIMD computers, but we do see it as a solution, particularly in the short run. We feel there is a role for an imperative language that is explicitly parallel, yet quite similar to existing sequential languages. An easy-to-learn data parallel language such as C* may help popularize parallel computing.

Acknowledgements This work was supported by National Science Foundation grants DCR-8514493, CCR-8814662, and CCR-8906622.

References