OREGON STATE
UNIVERSITY
COMPUTER
SCIENCE
DEPARTMENT

Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor

Michael J. Quinn
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

Philip J. Hatcher
Department of Computer Science
University of New Hampshire
Durham, NH 03824

89-60-20

Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor

Michael J. Quinn
Department of Computer Science
Oregon State University

Philip J. Hatcher
Department of Computer Science
University of New Hampshire

Abstract

Our thesis is that data parallel programs can be translated into programs that execute efficiently on a variety of architectures. Currently we are focusing on the data parallel programming language C*[TM], developed by Thinking Machines Corporation for the Connection Machine[TM] processor array. Previous papers have described our design of a cross-compiler for hypercube multicomputers. In this paper we propose the design of a cross-compiler that translates C* programs into C programs suitable for compilation and execution on a tightly coupled multiprocessor. The C* language is based upon a synchronous model of parallel computation, while the resulting C code executes asynchronously on the multiprocessor. We present an algorithm that, given a list of data dependences between expressions, produces a minimal set of barrier synchronizations necessary to ensure the correctness of the translated program.

1. Introduction

At a time when academics continue to debate the merits of various processor organizations and manufacturers continue to introduce novel parallel architectures, software portability is an important concern to parallel programmers. In the last few years a number of high level parallel programming languages and methodologies have been proposed, including The Force [Jordan 1987], Linda [Carriero and Gelernter 1989], Poker [Snyder 1984], Parallel Pascal [Reeves 1984], and C* [Rose and Steele 1986]. Because these languages are based upon abstract models of parallel computation, they are more machine independent and, hence, more portable.

Our work has focused on the data parallel programming language C*, designed by Thinking Machines Corporation for its Connection Machine processor array. Previous papers have described preliminary [Quinn et al. 1988] and refined [Quinn and Hatcher to appear] designs for a compiler that translates C* programs into C programs suitable for execution on the NCUBE family of multicomputers. These compiler-generated C programs often rival and occasionally match the speed of hand-coded C programs written for the NCUBE. In this paper we present the design of a C* compiler for tightly coupled multiprocessors. We focus on the issue of minimizing the number of synchronizations that must be added in order to maintain the semantics of the original C* program.

A *multiprocessor* is a multiple-CPU computer with a single address space [Quinn 1987]. Every processor can read from and write to every memory location. In a *tightly coupled multiprocessor* all processors work

---

C* and Connection Machine are trademarks of Thinking Machines Corporation. Balance is a trademark of Sequent Computer Systems.

through a central switching mechanism to reach a shared global memory. The tightly coupled multiprocessor used as a testbed in this paper is the Sequent Balance^{TM} 21000, which has a high speed bus as its central switching mechanism.

2. The Data Parallel Programming Language C*

We believe a synchronous, massively parallel model in which all the processes change state in a simple, predictable fashion leads to understandable programs. In this kind of model parallelism comes from the simultaneous execution of a single operation across a large data set.

The first advantage of the data parallel approach is its simple control flow. It is easy to determine the state of the processes, since all processes are either active or inactive as a universal program counter works through the various control statements. A second advantage is that results of computations are deterministic and independent of the number of physical processors used. A third advantage is that it is straightforward to build a debugger. Breakpoints make sense on a single instruction stream model. Lastly, data parallel programs are portable.

We want to emphasize that data parallel languages are not **the** solution to programming multiprocessors; they are only **a** solution. While many problems, such as low-level vision, protein mechanics, and most linear algebra problems, are amenable to solution via data parallel programs, a data parallel language is not appropriate for implementing programs with multiple asynchronous processes, such as data base management systems and multiprogrammed operating systems. More insight into the data parallel approach can be gained by reading the article by Hillis and Steele [1986].

The language C* is an extension of C that incorporates features of the data parallel programming model. We briefly describe a few important features of C*. For further details, see the paper by Rose and Steele [1986].

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++. 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.

```
domain vector { real 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 real 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 per 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 of the `temp` values of its predecessor and successor processing elements:

```
#define N 100
domain rod { real temp; } x[N];
< Intervening code >
[domain rod].{
  int index = this-x;  /* Meaning of this is same as in C++ */
  < Intervening code >
  if ((index > 0) && (index < N-1))
    temp = (x[index-1].temp + x[index+1].temp)/2.0;
}
```

Each processing element's value of `index` gives its unique position in the domain, a value in the range $0 \ldots N-1$. 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.

3. Evaluating the Cost of Synchronizations

Previous implementations of C* have been on architectures in which each processor has its own local memory. Processing elements are tightly bound to the physical processors emulating them, and when a processing element accesses a variable of a processing element mapped to a different physical processor, message passing is required. On a tightly coupled multiprocessor every processor has access to every memory location. The Balance C programming language allows variables to be declared as shared, meaning every

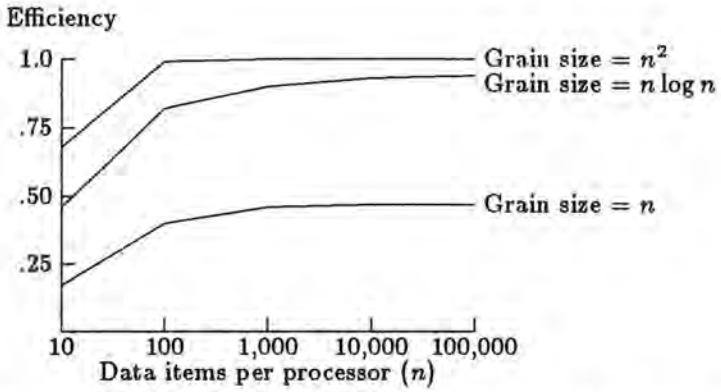

Figure 1. Maximum efficiency possible per processor as a function of number of data elements per processor and number of computations performed on these data elements.

CPU can read and write their values. Because the variables of all processing elements are accessible to all physical processors, this need for passing messages vanishes.

On the other hand, the values of variables in active domain instances must still be buffered at times. For example, consider the C* assignment statement

```
temp = (x[index-1].temp + x[index+1].temp)/2.0
```

taken from the second example in the previous section. Because the physical processors execute asynchronously on a multiprocessor, every processor must have access to the old `temp` values it needs before any processor assigns a new value. Either the old values of `temp` or the value of the expression must be buffered before the processors synchronize. After the synchronization the assignment can take place, using the values stored in the buffers.

In this section we evaluate the effect that this copying and synchronization overhead can have on the performance of the parallel program.

The maximum efficiency possible per processor is a function of both the number of data elements per processor and the number of computations performed on these data elements between synchronizations. Assume each processor's share of the data is n elements. Suppose at the heart of the parallel algorithm is an iteration in which each processor performs $f(n)$ operations on n elements, buffers n results, and then synchronizes with the other processors.

If K is the time spent per arithmetic operation, then the time needed to perform the computation is $Kf(n)$. Let p be the number of processors, S be the synchronization time per processor and C be the time needed to copy one data element into temporary storage. The overhead of the parallel program, then, is $Sp + 2Cn$, and the efficiency of the parallel program is

$$\text{Efficiency} = \frac{Kf(n)}{Kf(n) + Sp + 2Cn}$$

Reasonable estimates of the values of these constants on the Sequent Balance are $C = 23$ μsec, $K = 41$ μsec, and $S = 50$ μsec. Given these values, we have shown in Figure 1 the maximum efficiency possible, as a function of data set size, for $f(n) = n$, $f(n) = n \log n$, and $f(n) = n^2$ on a 30-processor system.

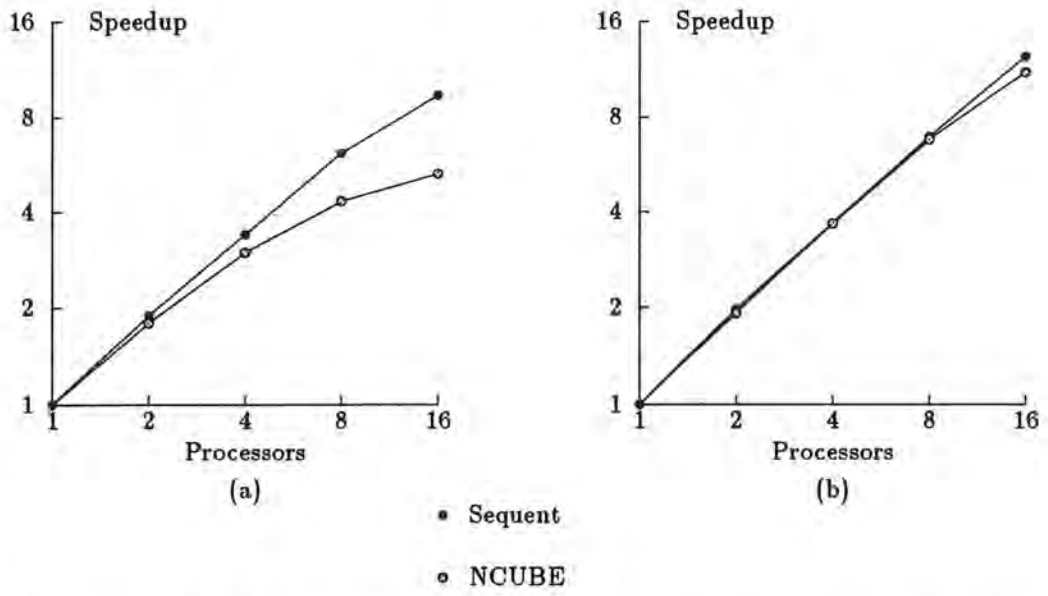

Figure 2. Performance of two parallel programs that decompose a dense system of linear equations into its LU factorization using Gaussian elimination. (a) Speedup achieved decomposing a system of size 64. (b) Speedup achieved decomposing a system of size 256.

This graph shows that for small-grained computations, buffering and synchronization time can dramatically lower the peak efficiency achievable by the parallel program. Hence it is vital that the number of synchronizations be minimized, especially inside loops.

To illustrate this concept with an actual example, we have hand-translated a C* Gaussian elimination program into a Sequent Balance C program and compared the speedup achieved by this program with the speedup achieved by a program executing on the NCUBE/7 hypercube multicomputer [Quinn and Hatcher to appear]. Figure 2 illustrates the results. Figure 2-a shows the speedups achieved on a system of 64 equations. The speedups vary widely, because the grain size is small enough that the time spent communicating has a large impact on speedup. The NCUBE/7 has lower speedup, since its synchronization and copying times are much higher. In Figure 2-b the computers are solving a system of 256 equations. The grain size is larger, so the time spent copying values is relatively less important, and the speedup curves are quite similar.

4. The Translation of C* Programs

Our compiler design must address two fundamental issues. The first issue is the insertion of synchronization points into the C program produced by the compiler so that (1) every processor participates in every barrier synchronization†, and (2) the semantics of the C program asynchronously executing on the multiprocessor are identical to the semantics of the original C* program. The second issue is the emulation

† We are making the simplifying assumption that every physical processor must participate in every synchronization. We believe there will be few instances in which this assumption dramatically lowers the potential performance of the parallel program, since a single physical processor emulates many virtual processors, and the exact set of processors that need to be involved in a synchronization is often dependent upon the input data.

of processing elements. Data parallel programs often assume a very large number of processing elements. If these programs are to run efficiently on a multiprocessor with far fewer physical processors, then there must be an efficient mechanism for emulating processing elements on physical processors.

4.1. Minimizing Number of Processor Synchronizations

The potential synchronization points of a C* program can be traced to those expressions in which processing elements read or write values to or from other processing elements. 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 synchronization 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. Standard techniques are then used to determine dependences between these expressions [Padua and Wolfe 1986].

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 every expression, every statement, or at the end of every block. However, if halfway through the block each processing element reads a value from its neighbor and that value may have been written after the last synchronization point, then the processors must be synchronized prior to accessing that value in shared memory, in order to guarantee that the value retrieved by each processor is the same as it would be under a fully synchronous implementation. Similarly, if a processing element writes a value into the memory of another processing element, and that memory location has been written into or read from by another processing element after the last synchronization, then a synchronization is required in order to preserve the semantics. Hence synchronization is required between expressions in which different processing elements access the same memory location.

A single block of code may have more than one data dependence requiring a synchronization. For example, consider the following code segment:

```
mono float csum;
domain foo { float a, b, c; } x[100];
<Intervening code>
[domain foo].{
  <Intervening code>
  csum += c;
  a = b + c;
  (this-1)->b = b;
  c = c / csum;
  <Intervening code>
}
```

Because active processing elements access in the fourth statement the value of csum, which is defined in the first statement, there must be a barrier synchronization between the first and the fourth statements. Because every active processing element overwrites in the third statement another processing element's variable, b, that is read in the second statement, there must be a barrier synchronization between the second and the third statements. A single barrier synchronization, placed between the second and the third

Given: List L of d pairs (i, j) representing expression ranges. The existence of pair (i, j) implies there must be a synchronization barrier placed immediately before at least one of the expressions in the range $i \ldots j$.
Result: Minimal set B of expressions immediately before which barrier synchronizations should be placed.

1. $B = \emptyset$
2. Sort list L by first value. Call resulting list F.
3. Sort list L by second value. Call resulting list S.
4. While list S is not empty do
   4a. $B = B \cup \{j\}$, where (i, j) is the first pair in S
   4b. While $j \geq k$, where (k, l) is the first pair in list F do
       Remove pair (k, l) from list F and list S

Figure 3. Algorithm *SelectSyncs*, which finds a minimal set of synchronization points, given a list of synchronization requirements, where all dependences are in forward direction.

Input:
L: (2,6), (3,4), (6,8), (2,5), (7,10), (4,6)

Sort list L into lists F and S:
F: (2,6), (2,5), (3,4), (4,6), (6,8), (7,10)
S: (3,4), (2,5), (4,6), (2,6), (6,8), (7,10)

Add {4} to B.
Remove (2,6),(2,5), (3,4), (4,6) from F and S.
F: (6,8), (7,10)
S: (6,8), (7,10)

Add {8} to B.
Remove (6,8), (7,10) from F and S.

Algorithm terminates. B = {4, 8}

Figure 4. Example of operation of algorithm *SelectSyncs*.

statements, is sufficient to achieve both purposes.

Adding Synchronization Points to Basic Blocks.

Fortunately, a simple greedy algorithm can extract a minimal set of synchronizations from a list of synchronization requirements in which every dependence is in the forward direction, from an earlier expression to a later expression. The algorithm, called *SelectSyncs*, is shown in Figure 3, and an example of its operation appears in Figure 4.

Complexity analysis. Let d be the number of data dependences in list L. Step 1 requires constant time. Steps 2 and 3 have complexity $\Theta(d \log d)$. Step 4a is executed at most d times during the course of the algorithm, and requires $O(d)$ time in all. The body of loop 4b is executed exactly d times during the course of the algorithm. Removing the first element from list F can be done in constant time. By doubly-linking list S and linking elements of F with the corresponding element in list S, the second deletion

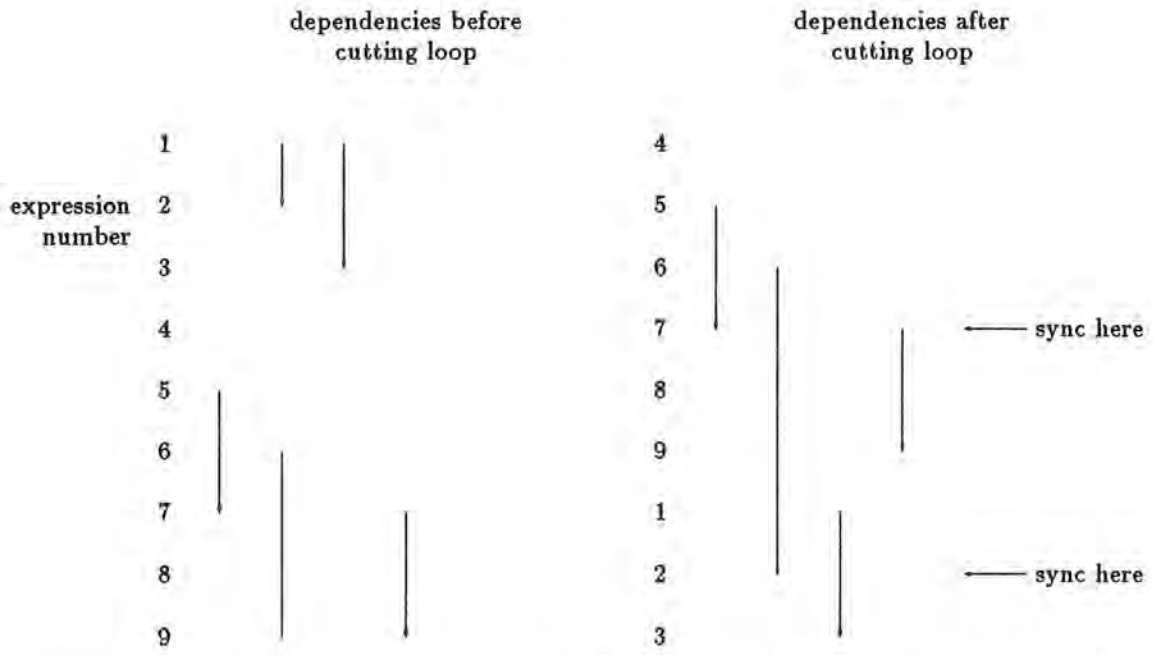

Figure 5. Determining synchronization points in a loop where data dependences do not overlap completely.

can be accomplished in constant time as well. Hence the total time complexity of step 4 of the algorithm is $\Theta(d)$. Algorithm *SelectSyncs* has time complexity $\Theta(d \log d)$.

Adding Synchronization Points to Loops.

Next we explain how to add synchronization points to a loop. If the loop contains only forward data dependences, it is handled using the basic *SelectSyncs* algorithm. If the loop contains backward data dependences, then there may or may not be an expression not spanned by a data dependence. If there is an expression not spanned by a data dependence, then the loop may be cut at that point and the basic algorithm applied to the resulting segment with "forward-only" dependences (see Figure 5). Given a loop with d data dependences and e expressions, it may require time $\Theta(e)$ to find an appropriate expression to cut the loop. Hence the complexity of this algorithm is $\Theta(e + d \log d)$.

If data dependences span every expression in the loop, then we must try cutting the loop at the end point of every data dependence. By putting a synchronization before that expression and removing all data dependences ensured by that synchronization, all remaining data dependences are "forward" and the elementary algorithm can be applied. After attempting to cut the loop at all such data dependence end points, we choose the solution requiring the fewest synchronizations. This algorithm has complexity $\Theta(e + d^2 \log d)$, where d is the number of data dependences and e is the number of expressions in the loop.

Midkiff [1986], Li and Abu-Sufah [1987], and Midkiff and Padua [1987] have presented algorithms for inserting synchronization points into the bodies of FORTRAN DO loops marked for parallel execution. Midkiff [1986] has shown that the problem of eliminating redundant dependences is NP-hard. We do not have to rely upon the concurrent execution of multiple loop iterations to achieve parallelism, since parallelism is explicit in the C* selection statement. The only kind of synchronization we are inserting is a barrier synchronization involving all physical processors. For these reasons our algorithm to insert a minimal set of synchronization points in a loop body is tractable.

The C* construct:

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

is translated into the following C code:

```
temp = TRUE;
do {
   if (temp) {
      temp = condition;
   }
   if (temp) {
      statement_list1;
   }
   gtemp = global_or (temp);
   barrier_synchronization;
   if (temp) {
      statement_list2;
   }
} while (gtemp);
```

Figure 6. Translation of C* `while` loop.

Adding Synchronization Points to an Entire Program.

Synchronization points are added to an entire program in the following manner. First the necessary synchronization points are added to the loops. The remaining data dependences are all in the forward direction. Those which span a synchronization point can be removed. Of those that remain, the loop portions of those that enter or leave loops are removed (since we do not wish to add any more synchronization points inside loops). Once these modifications have been made, algorithm *SelectSyncs* can be used to determine a minimal set of synchronization points.

Transforming Control Structures.

Because the barrier synchronization requires the 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 barrier synchronization. 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 compute a global logical *OR* of their Boolean loop control values. A processing element can exit the loop only when the global logical *OR* is *false*.

Of course, a processing element does not actually execute the body of the loop after its local loop control value has gone to *false*. Rather, the physical processor on which it resides participates in the barrier synchronization and the global *OR* operation. This means that our C* compiler must rewrite the control structure of input programs. Figure 6 illustrates how `while` loops are rewritten.

The requirement that all physical processors must actively participate in any barrier synchronization forces our compiler to rewrite all control statements that have inner statements requiring a synchronization. We must rewrite the statement in order to bring the synchronization to the surface of the control structure. Figure 7 illustrates how an `if` statement is handled.

Synchronizations buried inside nested control structures are pulled out of each enclosing structure until they reach the outermost level. Figure 8 shows a nested `if` statement.

The technique just described will not handle arbitrary control flow graphs. For this reason we have not

The C* construct:

```
if (condition) {
   statement_list1;
   barrier_synchronization;
   statement_list2;
}
```

is translated into the following C code:

```
temp = condition;
if (temp) {
   statement_list1;
}
barrier_synchronization;
if (temp) {
   statement_list2;
}
```

Figure 7. Translation of simple `if` statement.

The C* construct:

```
if (condition1) {
   statement_list_1;
   if (condition2) {
      statement_list2;
      barrier_synchronization;
      statement_list3;
   }
   statement_list4;
}
```

is translated into the following C code:

```
temp1 = condition1;
if (temp1) {
   statement_list1;
   temp2 = condition2;
   if (temp2) {
      statement_list2;
   }
}
barrier_synchronization;
if (temp1) {
   if (temp2) {
      statement_list3;
   }
   statement_list4;
}
```

Figure 8. Translation of nested `if` statements.

implemented the `goto` statement. We can, however, handle the `break` and `continue` statements.

4.2. Efficiently Emulating Processing Elements

Once the barrier synchronizations have been brought to the outermost level of the program, emulation of processing elements is straightforward. Every physical processor emulates an equal share of processing elements. The compiler puts `for` loops around the blocks of code that have been delimited by the synchronization steps. During each iteration of a `for` loop a physical processor emulates the actions of a different processing element. Since within the delimited blocks there is no interaction between processing elements, it makes no difference in which order the physical processors perform the operations of the processing elements.

4.3. Summary

To summarize, the compiler must first locate the expression ranges that must include a barrier synchronization. Second, the compiler must generate a minimal set of synchronizations that satisfy the synchronization requirements. Third, the compiler must transform the control structure of the input program to bring these synchronization steps to the outermost level. Finally, 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 barrier synchronizations.

5. Conclusions

We have examined some key issues in the translation of C* programs into semantically equivalent C programs suitable for compilation and execution on a tightly coupled multiprocessor. The existence of a shared global memory eliminates the need for communicating values of domain instances from one physical processor to another. However, copying values into temporary locations in global storage and synchronizing the physical processors are needed at times. We have examined how synchronizations can affect the efficiency of the parallel program, and we have presented an algorithm that minimizes the number of barrier synchronizations that must be introduced into the C program.

Acknowledgments

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

References

Carriero, N., and Gelernter, D. 1989. Linda in context. *Communications of the ACM* 32, 4 (April), pp. 444–458.

Goldberg, B., and Hudak, P. 1987. Alfalfa: distributed graph reduction on a hypercube multiprocessor. In *Graph Reduction: Proceedings of a Workshop*, Lecture Notes in Computer Science 279, Springer-Verlag, Heidelberg, Germany pp. 94–113.

Hillis, W. D., and Steele, G. L., Jr. 1986. Data parallel algorithms. *Communications of the ACM* 29, 12 (December), pp. 1170–1183.

Hudak, P., and Goldberg, B. 1985. Efficient distributed evaluation of functional programs using serial combinators. In *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 831–839.

Hudak, P., and Goldberg, B. 1985. Serial combinators: "Optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer-Verlag, Heidelberg, Germany pp. 382–388.

Jordan, H. 1987. The Force. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds., The MIT Press, Cambridge, MA, pp. 395–436.

Li, Z., and Abu-Sufah, W. 1987. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers* C-36, 1 (January), pp. 105–109.

Midkiff, S. P. 1986. Automatic generation of synchronization instructions for parallel processors. MS thesis, University of Illinois at Urbana-Champaign, CSRD Report 588 (May).

Midkiff, S. P., and Padua, D. A. 1987. Compiler algorithms for synchronization. *IEEE Transactions on Computers* C-36, 12 (December), pp. 1485–1495.

Padua, D. A., and Wolfe, M. J. 1986. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29, 12 (December), pp. 1184–1201.

Quinn, M. J. 1987. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, NY.

Quinn, M. J., and Hatcher, P. J. To appear. Data parallel programming on multicomputers. *IEEE Software.*

Quinn, M. J., Hatcher, P. J., and Jourdenais, K. C. 1988. Compiling C* programs for a hypercube multicomputer. In *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988, *SIGPLAN Notices* 23, 9 (September), pp. 57–65.

Reeves, A. P. 1984. Parallel Pascal: An extended Pascal for parallel computers. *Journal of Parallel and Distributed Computing* 1, pp. 64–80.

Rose, J. R., and Steele, G. L., Jr. 1986. C*: An extended C language for data parallel programming. Tech. report PL 87–5, Thinking Machines Corporation, Cambridge, MA.

Snyder, L. 1984. Parallel programming and the Poker programming environment. *Computer* 17, 7 (July), pp. 27–36.