Synchronization Issues in Data-Parallel Languages*

Sundeep Prakash Maneesh Dhagat Rajive Bagrodia
Computer Science Department
University of California
Los Angeles, CA 90024

February 10, 1994

Abstract

Data-parallel programming has established itself as the preferred way of programming a large class of scientific applications. In this paper, we address the issue of reducing synchronization costs when implementing a data-parallel language on an asynchronous architecture. The synchronization issue is addressed from two perspectives: first, we describe language constructs that allow the programmer to specify that different parts of a data-parallel program be synchronized at different levels of granularity. Secondly, we show how existing tools and algorithms for data dependency analysis can be used by the compiler to both reduce the number of barriers and to replace global barriers by cheaper clustered synchronizations. Although the techniques presented in the paper are general purpose, we describe them in the context of a data-parallel language called UC developed at UCLA. Reducing the number of barriers improves program execution time by reducing synchronization time and also processor stall times.

1 Introduction

Data-parallel programming has established itself as the preferred way of programming a large class of scientific applications. The primary distinction between a data-parallel program and a control (or task) parallel program is that the former has a single locus of control whereas the latter typically has multiple locii of control. This feature makes it considerably easier to design and debug data parallel programs.

A number of data-parallel languages have been designed and implemented. In most exisiting language proposals, the granularity of synchronization is determined by the specific constructs provided by the language. For instance, at one extreme we have essentially SIMD languages like C∗ [RS87] and Dataparallel C [HQL+91] where synchronization is defined at the level of individual

*This research was partially supported under NSF PYI Award No. ASC-9157610, ONR Grant No. N00014-91-J-1605, and Rockwell International Award No. L911014.

operations: all parallel threads are either executing the same operation or are idle. On the other hand, languages like Kali [MR90] restrict access to remote data only at selected points in a program, like at the beginning of successive iterations of a *forall* loop. This ensures that a Kali program needs to be synchronized only at precisely those points where communication is possible; the multiple threads can execute asynchronously between two successive synchronization points. Other languages like UC [ABCD93] and DINO [RSW91] permit synchronization with multiple grain size: for instance parallel functions use copy in/copy out semantics so that they need to be synchronized only at the point of call and return; other statements are synchronized at a finer level of granularity.

The synchronization granularity of a language impacts both its semantics and the efficiency of its implementations on asynchronous architectures. Thus, languages that are synchronized at the operation level have simple semantics, as the programmer has access to the correct global state of the program before executing any operation. However, implementing such a strict notation on an asynchronous distributed memory architecture is expensive as a barrier may potentially be required before the execution of every operation[1]. This is particularly the case if the language uses a universal shared memory model where no syntactic distinction is made between local and remote data. In contrast, a language with a weaker synchronization model forces the programmer to restrict remote data access to specific points in their code. This places the burden of maintaining a coherent view of the shared data explicitly on the programmer. However, an implementation of this model on an asynchronous architecture requires a smaller number of barrier synchronizations as compared with the previous model. Reducing barrier synchronizations improves execution efficiency by reducing both the blocking and communication times.

In this paper, we address the issue of reducing synchronization costs when implementing a data-parallel language on an asynchronous architecture. The synchronization issue is addressed from two perspectives: first, we describe language constructs that allow the programmer to specify that different parts of a data-parallel program be synchronized at different levels of granularity. This allows the programmer to take on the burden of explicit data management to reduce synchronization costs. It also supports an iterative approach to program design where programs are initially designed assuming operation level synchronization and may be later refined to use a weaker synchronization granularity. Secondly, we show how existing tools and algorithms for data dependency analysis can be used by the compiler to both reduce the number of barriers and to replace global barriers by cheaper clustered synchronizations. The next section is a brief description of the language constructs. Particular emphasis has been placed on the constructs that are used to alter the synchronization granularity of the programs. Section 3 describes the code transformations used by the compiler to either remove unnecessary barriers or replace them by less expensive operations. Section 4 presents a preliminary performance analysis of the compiler transformations. Section 5 is the conclusion.

[1]The issue of compiler optimizations and data distributions that reduce the number of synchronizations is addressed subsequently.

```
1 int a[N][N], b[N][N], c[N][N];
2 index_set I:i = {0..N-1}, J:j=I, K:k=I;
3 par(I,J)
4     c[i][j] = $+(K; a[i][k]*b[k][j]);
```

Figure 1: Matrix multiplication in UC

2 The UC Programming Language

UC [ABCD93] is a data-parallel extension of C. This section gives a brief description of the language. For a complete description, the reader is referred to [BA92].

UC enhances C with a data-type called *index-set* and some new operators to specify parallel execution of statements. An index-set represents an ordered set of integers. The most commonly used UC operators are *reduction* and *parallel assignment*. Reduction is used to perform an associative, commutative binary operation on a set of elements; a parallel assignment is used to simultaneously modify a set of elements. The set of elements used by these parallel operators is specified by an *index-set*. In addition, an optional predicate allows the operation to be performed on a subset of elements.

Consider the UC program in Figure 1. The program computes matrix c as the product of two $N \times N$ matrices a and b. The program declarations include the standard C declarations for matrices a, b and c (line 1). In addition, three index-sets I, J and K are declared (line 2). The declaration of an index-set uses two identifiers, the first one (for example I) refers to the index-set itself and the second (for example i) refers to an arbitrary element of the set. Each index-set in the example consists of N integers from 0 to N-1.

The UC keyword **par** (line 3) specifies parallel execution of the assignment statement in line 4. The index-set(s) contained in the par statement determine the number of instances of the assignment statement that are executed in parallel. In this case, N^2 instances of the assignment will be executed, one for each of the N^2 elements in the product set $I*J$. Now consider the assignment statement itself. For a given (i, j), the value of *c[i][j]* is computed as the dot product of two vectors: the i^{th} row of a and the j^{th} column of b. The dot product is programmed in UC as a reduction (line 4). The reduction (denoted by '**$+**') specifies the addition of N expressions, one for each value of index-element k, where the k^{th} expression is *a[i][k]*b[k][j]*. Besides addition, other associative, symmetric operators may also be used in a reduction. These include '**$&&**' for logical AND, '**$||**' for logical OR, '**$>**' for maximum, '**$<**' for minimum, '**$***' for multiplication and '**$,**' to return the value of an arbitrary operand.

The next example illustrates the use of a mask to select a subset of the elements from an index-set. The program in Figure 2 computes the trace of array a (the sum of elements on its main diagonal). The declarations are similar to the previous example and are omitted. The body is a single reduction which computes the sum of the diagonal elements. The predicate $(i == j)$ selects the diagonal elements of array a. The keyword **st** (such that) in the reduction separates

```
trace = $+(I,J st (i == j) a[i][j]);
```

Figure 2: Trace computation in UC

the index-sets from the boolean expression. (Note that this reduction may be expressed more concisely as: **$+***(I; a[i][i])*. If the operand list of a reduction is empty, the reduction returns the identity value for the corresponding reduction operator. A boolean expression may also be used in a par statement to restrict the assignment operation to a subset of the array elements. The par statement may also contain a sequence of statements. In this case, successive statements are executed synchronously for each element in the index-set(s) named in the enclosing par statement. If a par statement includes (parallel) function calls, multiple instances of the called function are executed asynchronously. Function parameters are passed by value result and output parameters of the parallel function are required to be non-interfering, i.e. an actual parameter may be modified by at most one instance of the parallel function. Global access in a parallel function is restricted to read only data. The mask in a par statement can be an arbitrary predicate and the par statement may contain any UC or C statements including conditionals, iterative construct or nested par statements.

A par statement specifies synchronous execution of a a set of statements, where the multiple instances are synchronized at the operation level. The keyword **par** may be thought of as a composition operator that specifies synchronous execution of its statement operands. A number of other composition operators are also supported by UC. These include **seq** for sequential execution, **arb** for asynchronous execution, and ∗**solve** for fixed-point computations. We describe the **seq** and **arb** composition operators.

If the keyword **par** of a UC statement is replaced by **seq**, the corresponding statement (or statement block) is executed sequentially for each element in the index-set. The elements are selected in the order specified in the declaration of the corresponding index-set. If a seq statement includes multiple index-sets then the order of the elements in the product set is determined by the order in which the index-sets are specified in the statement. For example, consider the UC program of Figure 1 with keyword par in line 3 replaced by seq. This modification will cause elements in matrix c to be computed sequentially in row major order.

If the keyword **par** of a UC statement is replaced by **arb**, the multiple instances of the corresponding statement are assumed to be independent and may hence be executed asynchronously. The arb statement may be implemented more efficiently than the par statement on asynchronous architectures. To preserve the synchronous semantics of a par statement on an asynchronous architecture, a barrier synchronization is needed prior to the evaluation of each operation in the statement. Of course, a smart compiler may be able to analyze the dependencies in the statement and remove unnecessary barriers. However, for many statements, particularly those where array subscript expressions include variables or array references, the analysis is hard and the compiler may be forced to insert barriers in a conservative manner. The arb statement may be used by a programmer to specify a weaker granularity of synchronization for the program. Consider the following code fragments:

```
par (I) {                    arb (I) {
```

```
    a[f(i)] = b[i];                         a[f(i)] = b[i];
    c[i] = a[g(i)];                         c[i] = a[g(i)];
}                                       }
```

When compiling the fragment on the left, the compiler must place a barrier between the two statements for consistency with the semantics of a par statement. However, if the programmer can infer that the two statements access non-overlapping sets of elements from array *a*, the par statement may be replaced by the arb statement on the right, rendering the barrier unnecessary. Arb and par statements may be nested within each other to modify the synchronization as needed. In the following example, the arb statement is used to specify that functions *f* and *g* reference non-overlapping elements of *a*, which allows the statement to be executed asynchronously – thus reducing the number of barrier synchronization. Note that an optimizing compiler could easily deduce that the first assignment to array *a* may also be moved within the arb statement, although making this determination for the third assignment statement is harder. Compiler optimizations are discussed in the next section.

```
par (I) {
    a[i] = h(i);
    arb
        a[f(i)] = b[i] + a[g(i)];
    c[i] = a[h(i)];
}
```

3 Multicomputer Implementation

The main issues in implementing a dataparallel language on a multicomputer are (1) Data mapping and (2) Synchronization detection. In this paper, we concentrate on the second issue. We use standard data dependency analysis to reveal and efficiently implement the minimum number of synchronization points needed to correctly implement the statements of a dataparallel language. We use the par statement of UC as an example.

The synchronous semantics of a par statement may be obtained by a straightforward implementation that introduces a barrier before evaluating every parallel operand. However, for almost any program, the number of barriers can be dramatically reduced by examining the data dependencies among variables referenced and updated in the statement.

We first show how well-known data dependence analysis techniques can be used to construct a *synchronization graph* for a par statement. This graph can be used to deduce the minimal number of barriers needed to preserve program dependencies. We then show how additional information from the synchronization graph can be used to replace some barriers with less expensive pair-wise synchronization operations. There are obvious similarities between our goals and the use of dependency graphs for concurrentization of sequential programs[PW86]. However, there are some important differences: whereas concurrentization requires that all analysis be completed at compile-time, we are able to do some run-time analysis to further refine dependency information. Secondly, we use dependency information to replace barriers by fuzzy barriers[Gup89] which allow a thread to conditionally continue execution beyond the global barrier, before the other threads have reached the barrier. Although analysis is presented here in the context of a UC program, the results and techniques are equally applicable for other data parallel languages including HPF[For92].

In the remainder of this section, we assume that each array has been folded into a number of uniform segments, where each segment is mapped to a unique processor.

3.1 Synchronization Graphs

Here we briefly describe the three types of data dependencies that occur in UC programs. The description is adapted from [BENP93] where the dependencies were introduced in the context of for loops in sequential programs. In this section, these dependencies are defined in the context of par statements and it is shown how they can be easily extracted and used to construct synchronization graphs for UC programs. The following small example illustrates the three types of data dependencies:

```
par (I)
{
    a[i] = c[i];                  /* s1 */
    b[i] = a[i+1] + c[i];         /* s2 */
    b[i] += d[f(i)];              /* s3 */
    d[i] = c[i];                  /* s4 */
}
```

The notation s_j denotes the instance of statement s for an element j of the index-set I.

- **Flow** dependence: the value computed by the instance $s1_j$ is used by the instance $s2_{j-1}$. $s2$ is said to be flow dependent on $s1$, or $s1\ \delta^{\mathrm{f}}\ s2$.
- **Anti**-dependence: a value for variable d[f(j)] is used by the instance $s3_j$, and another value for this variable is later computed by instance $s4_{\mathrm{f}(j)}$. $s4$ is anti-dependent on $s3$, or $s3\ \delta^{\mathrm{a}}\ s4$.
- **Output** dependence: the instance $s2_j$ and the instance $s3_j$ both compute a value for the variable b[j], such that the value computed by $s3_j$ is stored after the value computed by $s2_j$. $s3$ is output dependent on $s2$ or $s2\ \delta^{\mathrm{o}}\ s3$.

Note that, in using par, we are working with an explicitly parallel notation. If the body of the above par statement existed instead in a *do-loop* (or, equivalently, a UC seq loop), the dependencies would become the following (where the notation s_j denotes that the instance of statement s for an iteration j): (1) $s2_j\ \delta^{\mathrm{o}}\ s3_j$, (2) $s2_j\ \delta^{\mathrm{a}}\ s1_{j+1}$, and (3) $s3_j\ \delta\ s4_{\mathrm{f}(j)}$[2].

A synchronization graph for a par statement may be obtained by simple transformations on a data dependency graph derived for a corresponding *do loop*. The do loop is derived as follows: the

[2]May be a flow or anti-dependence, depending on the nature of the expression f(j)

body of the loop is the same as the body of the par statement and the iteration range is defined by the bounds of the index-set. The dependency graph for the loop can be derived using well-known techniques presented in standard references on dependency analysis [AK87, Wol89, FOW87, BENP93]. The transformation rules for deriving a synchronization flow graph from a dependency graph are summarized below:

- Flow and anti-dependencies within the same statement both translate to anti-dependencies. Consider the following statement:

  ```
  a[e(i)] = ... a[f(i)] ...;
  ```

  If *e(i)* were *i* and *f(i)* were *i-1* this would be flow dependency in a do loop. If *f(i)* were *i+1* this would be an anti-dependency. In a par statement both are anti-dependencies.

- All flow and anti-dependencies in which the statement using a variable occurs before the statement defining it translate to antidependencies. Consider the following statements:

  ```
  ... = ... a[f(i)] ...;        /* s1 */
  a[g(i)] = ...;                /* s2 */
  ```

  If *f(i)* were *i-1* and *g(i)* were *i* there would be a flow dependency from *s2* to *s1* in a do loop. If *f(i)* were *i+1* there would be an anti-dependency from *s1* to *s2*. In a par statement both are anti-dependencies from *s1* to *s2*.

- All flow and anti-dependencies in which the statement defining a variable occurs before the statement using it, translate to flow dependencies. Consider the following statements:

  ```
  a[g(i)] = ...;                /* s1 */
  ... = ...a[f(i)]...;          /* s2 */
  ```

  There could either be a flow dependency from *s2* to *s1* or an anti-dependency from *s1* to *s2* in a do loop. In a par statement there will be a flow dependency from *s1* to *s2*.

- Output dependencies remain output dependencies in all cases.

In concurrentizing a do-loop, we distinguish between inter-iteration and intra-iteration dependencies. In the context of par statements, we similarly distinguish between *inter-thread* and *intra-thread* dependencies. Intra-thread dependencies can be deleted from the synchronization graph, since a thread executes sequentially, such dependencies are automatically obeyed.

The synchronization graph for the par statement shown in the beginning of the section appears in figure 3(a). We now show how some simple program transformations can eliminate some of the dependencies in the synchronization graph.

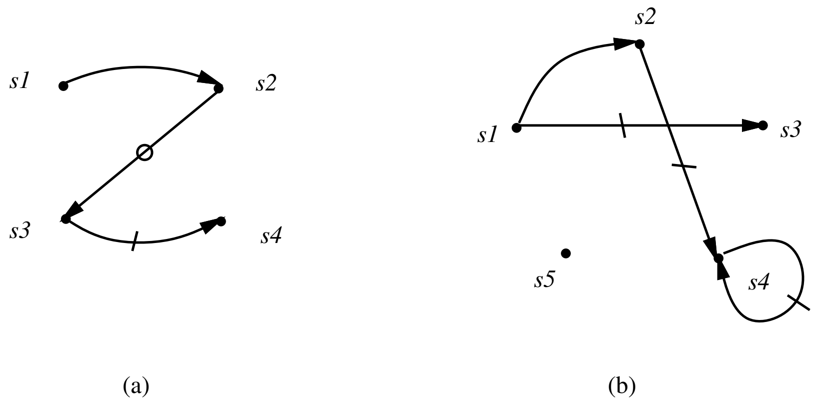

Figure 3: Synchronization Graphs

```
par (I) {
    a[i+1] = a[i] + 1;      /* s1 */
    b[i] = a[i] + c[i];     /* s2 */
    a[i] = d[i] + e[i];     /* s3 */
}
```

```
par (I) {
    tmp[i+1] = a[i] + 1;    /* s1 */
    b[i] = tmp[i] + c[i];   /* s2 */
    a[i] = d[i] + e[i];     /* s3 */
}
```

Figure 4: Using Renaming to remove Output and Anti-Dependencies

3.2 Program transformations

As shown in [PW86], simple program transformations can eliminate some dependencies in do loops. Many of the same transformations can be applied on par statements to eliminate dependencies in the synchronization graph. In particular, we show how *renaming* and *alignment* can be used to remove some dependencies.

The program fragment in figure 4 (on left) exhibits the inter-thread dependencies: (1) *s1* δ^a *s1*, (2) *s1* δ^o *s3*, and (3) *s1* δ^f *s2*. The first two dependencies can be eliminated by renaming the array a as shown in the transformed program on the right.

Alignment or loop skewing [MA87] is another technique to remove inter-thread dependencies. By selectively shifting the access pattern of some statements in the par statement body it is possible to reduce the number of dependencies. Consider the code fragment in figure 5 (on left) which exhibits the dependencies: (1)*s1* δ^f *s2*, (2) *s2* δ^a *s3* and (3) *s1* δ^o *s3*. Shifting the access patterns of *s1* and *s3* results in the transformed code on the right where no inter-thread dependencies exist. This transformed par statement can now be re-written as an arb statement, which does not require any synchronization.

```
index_set I:i = {0..N};
par (I) {
    a[i+1] = 2 * c[i];      /* s1 */
    b[i] = a[i] + c[i];     /* s2 */
    a[i+2] = d[i] + e[i];   /* s3 */
}
```

```
index_set I:i = {0..N+2};
par (I) {
    if ((i >= 1) && (i <= N+1))
        a[i] = 2 * c[i-1];          /* s1 */
    if ((i >= 0) && (i <= N))
        b[i] = a[i] + c[i];         /* s2 */
    if ((i >= 2) && (i <= N+2))
        a[i] = d[i-2] + e[i-2];     /* s3 */
}
```

Figure 5: Using Alignment to remove Dependencies

3.3 Minimizing Number of Barrier Synchronizations

The operation-by-operation synchronization model persented thus far is overly restrictive. Given a synchronization graph, it is possible to deduce the minimal number of barriers required such that each dependency is enforced. More precisely, given a set of intervals spanned by flow, anti, and output dependencies, the minimal number of barrier sychronizations must be determined such that each interval spans at least one barrier. This problem can be solved in polynomial time[Gav72]; a simple greedy algorithm devised by Quinn and Hatcher[QHS91] extracts a minimal set of synchronizations from a set of dependencies arcs represented as a sychronization graph. At each step the algorithm selects, of all arcs, the dependency arc whose head points to the earliest point in the program. A barrier is inserted before this point and the arc is removed from the synchronization graph. Additionally, all arcs whose tail is at a point before and the head at a point after the barrier are removed from the synchronization graph. The algorithm terminates when the synchronization graph contains no more arcs.

The example in figure 6 (on left) illustrates the use of this algorithm. The dependency arcs are shown in the synchronization graph of figure 3(b). The algorithm determines that two barriers are needed: before statement *s2*, and before the assignment in *s4*. Finally, the equivalent code with minimum number of barriers implemented with UC arb statements is shown in figure 6 (on right).

3.4 Conditions for Weakening Barriers

On an asynchronous distributed memory machine, implementation of an inter-thread dependency requires a set of communication operations between pairs (or clusters) of processors. Barriers perform only a part of this function by synchronizing all processors. Processors must, at run-time, request and obtain data from each other to complete the communication operation. This implies that processors must block at two points (1) at the barrier (2) while waiting for the requested data. In many cases, it is possible to identify the communication pattern at compile time, which enables processors to send data to other processors without having to wait for a request. This not only eliminates barriers, but reduces the time a processor waits for a data item to arrive.

```
par (I)
{
    a[i] = b[i] + c[i];          /* s1 */
    d[i] = a[i+1] + e[i];        /* s2 */
    b[i+1] = m[i] + g[i];        /* s3 */
    e[i-1] = e[i+1] + d[i];      /* s4 */
    m[i] = h[i] + j[i];          /* s5 */
}
```

```
arb (I) {
    a[i] = b[i] + c[i];
}
arb (I) {
    d[i] = a[i+1] + e[i];
    b[i+1] = m[i] + g[i];
    tmp[i] = e[i+1] + d[i];
}
arb (I) {
    e[i-1] = tmp[i];
    m[i] = h[i] + j[i];
}
```

Figure 6: Deducing Minimal Number of Barriers

Consider the following code fragment which shows how a flow dependency may be implemented by a communication operation. We assume that all the statements are aligned, hence every flow dependency must be of the form shown:

```
par (I) {
    a[i] = ... ;                 /* s1 */
    ...
    b[i] = a[g(i)];              /* s2 */
}
```

Here, *a[i]* is written by processor i and later used by processor $g^{-1}(i)$. If $g^{-1}(i)$ can be calculated at compile time, the par statement may be rewritten as shown below. We use the notation **send**(a, {i}) to mean "send a message containing the value of the locally stored variable a to the set of processors i." Similarly, the notation **blx_recv**(b, i) means "block until a message is received from processor i, and assign the value contained in the message to the locally stored variable b." For simplicity, we assume that the order of messages is preserved.

```
par (I) {
    a[i] = ... ;                 /* s1 */
    send(a, {g^-1(i))});
    ...
    blx_recv(tmp, g(i));
    b[i] = tmp;                  /* s2 */
}
```

The code for sending *a[i]* to processor $g^{-1}(i)$ should be moved up as much as possible in order to hide the communication latency.

Anti-dependencies can be handled in much the same way. Consider the above code fragment with the statements *s1* and *s2* interchanged. The antidependency requires processor i to transmit the value of *a[i]* to processor $g^{-1}(i)$ before it writes it at statement *s1*. In fact, for every anti-dependency, there must exist a flow dependency before it. Implementing the flow dependency by a communication operation automatically eliminates the need for the implementation of the anti-dependency.

An example clearly shows the working of this method: The following **par** statement

```
par (I) {
    a[i] = b[i+1] + c[i];          /* s1 */
    f(c[i]);
    b[i] = a[i+1] / 2;             /* s2 */
}
```

contains the dependencies *s1* δ^f *s2* and *s1* δ^a *s2*. The following is the transformed program:

```
arb (I) {
    if (i > 0) send(b, i-1);
    if (i < N-1) blx_recv(a, i+1);
    a[i] += c[i];                  /* s1 */
    if (i < N-1) send(a, i+1);
    f(c[i]);
    if (i > 0) blx_recv(b, i-1);
    b[i] /= 2;                     /* s2 */
}
```

As stated earlier, we have assumed that all statements are aligned. This automatically eliminates output dependencies. However, not all statements can be aligned. A statement of the following form may only be aligned if $g^{-1}(i)$ can be calculated at compile time:

```
par (I)
    a[g(i)] = ... ;
```

In general, whenever $g^{-1}(i)$ can be calculated (for all dependencies), a barrier can be replaced by significantly cheaper communication operations. In other cases, a barrier may be used. However, some run-time methods may be used to eliminate a barrier in some of these cases. Such cases are explored in subsequent sections.

3.5 Run Time Methods

3.5.1 Fuzzy Barriers and Non-blocking Requests

If communicating pairs (or clusters) between two dependent statements cannot be identified at compile-time, the compiler is forced to insert a barrier to maintain the semantics of the par statement. However, the total blocking time may be reduced by time taken for the computation in between the two statements, either by using fuzzy barriers or issuing early requests for data. Consider the following code segment:

```
par (I) {
    a[i] = ... ;            /* s1 */
    ...
    b[i] = a[g(i)];         /* s2 */
    ...
    a[i] = ... ;            /* s3 */
}
```

If the function $g^{-1}(i)$ is unkown at compile-time, a barrier must be inserted between *s1* and *s2*; and between *s2* and *s3*. Also, a request must be generated at *s2* for data element *a[g(i)]*. The two blocking points are (1) at the barrier and (2) at the data request point. The time at the blocking points may be reduced by two methods: *fuzzy barriers* and *non-blocking requests*.

The blocking time at the barrier can be reduced by replacing it with a fuzzy barrier[Gup89] extending from just after *s1* to just before *s2*. No thread may proceed to *s2* till all thread have finished executing *s1*. This reduces blocking time at the barrier by overlapping it with the computation performed between *s1* and *s2*. Fuzzy barriers can be implemented in hardware[Gup89], or in software in much the same way as a barrier. Each processor broadcasts to all other processors on reaching the beginning of the fuzzy barrier, continues computation until the end, and then waits until it has collected the broadcast messages of all other processors.

The blocking time at request for remote data can be reduced by inserting a barrier just after *s1* and making a non-blocking request for *a[g(i)]*; the value is later received by a blocking receive operation at *s2*. This reduces the communication latency by overlapping with the computation performed between *s1* and *s2*.

Both methods, however, cannot be applied simultaneously. In case where processors are synchronized to a greater degree, the second method may be preferable, and in cases where processors are loosely synchronized (and waiting time at the barrier is high) the first method may perform better.

The symmetrical case to the above is the following:

```
par (I) {
    a[g(i)] = ... ;         /* s1 */
    ...
    b[i] = a[i];            /* s2 */
}
```

In this case, it is not known (and cannot be locally calculated) which thread will write the value of *a[i]* at *s1*, which is then to be used at *s2*. However, a fuzzy barrier between statements *s1* and *s2* guarantees that when thread i reaches *s2* all threads have completed *s1*, hence ensuring that the value of *a[i]* read is the most recent one.

3.5.2 Definition Variables

In the first case considered in the previous section, the reason a barrier is needed is to guarantee that all requests for data have been serviced before the data is changed. The requests cannot be anticipated at compile time. An obvious solution is to make a copy of the data at the point its value is initialized. This is the function performed by a definition variable. Requests for the value of a definition variable wait for service until the variable gets initialized. The blocking time of the barrier is thus eliminated, leaving only the blocking time waiting for the request to be serviced. There is also the run-time overhead of implementing the definition variable, which is as yet unmeasured.

Consider the first code fragment of the previous section. If implemented without any barriers, the data request for *a[g(i)]* is made as early as possible before statement *s2*. However, if this request arrives at processor *g(i)* (1) before it has executed statement s1 or (2) after it has executed statement *s3*, a race condition is generated. To prevent this we use a *definition variable* which is assigned the value of *a[i]* at statement *s1*. Consider the following transformed code fragment (where **request** is defined similarly as send):

```
par (I) {
    a[i] = ...;                    /* s1 */
    def_a[i] = a[i];               /* s1' */
    request(def_a, g(i));          /* s1" */
    ...
    blx_recv(tmp, g(i));           /* s2' */
    b[i] = tmp[i];                 /* s2 */
    ...
    a[i] = ...;                    /* s3 */
}
```

In the above code, $def_a[i]$ is assigned at statement *s1'*. Any requests received before that are queued at the variable. The moment it is assigned, all waiting requests are serviced. The definition variable can be freed after all processors get past statement *s2* as no more requests for the value of $def_a[i]$ will be received. The life of the definition variable extends from *s1'* to a barrier after *s2* referred to as the *garbage collection barrier*. No processor actually needs to wait at a garbage collection barrier; it is merely used to detect the point at which we may free the memory used for the definition variable. Using this method, the processor only wait which its data request is being serviced.

The same method of definition variables cannot be used for the second code fragment of the previous section. If we make *a[i]* a definition variable at *s1*, then thread i would wait at statement *s2* for some other thread to define the value of *a[i]*. This may never happen as $g(j) = i$ may not have a solution for $j \in I$. A fuzzy barrier is the only solution to such a case.

3.6 Parallel Assignment with Conditionals

If only barrier synchronization is used, it is sufficient to perform a data dependency analysis on a par statement ignoring conditionals (and conservatively put barriers). However, when substituting these barriers with pairwise or clustered synchronizations, we must additionally take into consideration that only a subset of the processors may actually need to transmit values, since only a subset of processors may be executing matching receives. Consider the example:

```
par (I) st (e[i] < f[i]) {
    a[i] = b[i+1] + c[i];          /* s1 */
    b[i] = (a[i] + a[i+1])/2;      /* s2 */
}
```

Consider the following transformed code where the guard in the par statement is evaluated and saved in a temporary array called *guard*:

```
par (I) {
    if (guard[i-1])    send (b, i-1);
    if (guard[i]) {    blx_recv(tmp, i+1);
                       a[i] = tmp[i] + c[i];    }
    if (guard[i-1])    send(a, i-1);
    if (guard[i]) {    blx_recv(tmp, i+1);
                       b[i] = (a[i] + tmp[i])/2;    }
}
```

This assumes that the guard for the $(i-1)^{th}$ processor is available to the i^{th} processor. In general, whenever send-receive pairs are used, the guards can also be communicated using send-receive pairs. The code to communicate the guards can be inserted immediately after the point the guards are evaluated.

A par statement that contains a nested if statement is handled similarly.

3.7 Loops inside par statements

In general, data dependency analysis for statements within a loop tends to be more complex. To isolate the loop from the enclosing code, we put a barrier before and after the loop body. A synchronization graph is then built for the statements within the loop. Standard data dependency analysis for sequential do loops can be used to determine the flow, output and anti-dependencies. In following code fragment,

```
par (I)
    seq (J) {
        ...
        a[i][j] = ...;
        ...
```

```
        ... = a[f(i,j)][g(i,j)];
    }
```

dependencies that occur across index-set J are intra-thread dependencies and can be ignored for the purposes of synchronization. Only dependencies that occur only across index-set I are significant and can be renamed according to the rules mentioned before. The dependencies that occur across both the index-sets (I and J) remain unchanged.

In this manner, a synchronization graph can be obtained for the statements within a loop in a par statement. The dependencies can now be handled in the same way as before, except the arcs in the synchronization graph now represent J communication operations per thread rather than a single operation. In the case where we need definition variables, J separate definition variables must be used, one per communication operation.

4 Performance Estimation

This section presents a preliminary performance study of the effectiveness of the optimizations. We estimate the blocking time due to barriers and send-receive pairs and use the formula to estimate the improvements for a UC program fragment from a fluid dynamics application. The implementation of the compiler is in progress.

We use the term *block time* to refer to the stall time of a processor (we assume that a thread is assigned to each processor). A barrier synchronization involves all processors. Assume t_i is the time thread i arrives at the barrier, and t_{last} is the time the final thread arrives at barrier. Assuming N processors are involved in a synchronization, the average block time at the barrier is given by

$$T_{bar} = \frac{\sum_i (t_{last} - t_i)}{N}$$

A send-receive pair (or binary synchronization) involves two processors. To estimate average block time we use the average over the time any processor waits for any other processor. This is represented as a difference in arrival times as follows:

$$T_{sr} = \frac{\sum_j \frac{\sum_i |t_j - t_i|}{N-1}}{N}$$

Note that we have assumed that any pair of processors are equally likely to communicate[3]. A number of synthetic benchmarks were executed assuming various distributions for the process arrival time at the barrier. In particular, T_{bar} and T_{sr} were computed for 3 different distributions.

[3]However, in reality, communication may display locality, with processors communicating more often with neighbors.

distribution	T_{bar}	T_{sr}
uniform	50.016	32.881
normal	64.997	16.751
exponential	64.767	16.736

Figure 7: Average block times under various distributions

For N = 100 (λ = 0.5, mean = 50), these values are shown in Figure 7. In addition to the higher average block time, the overhead for barrier implementations are generally higher than for point-to-point communication.

In implementing UC programs, we distinguish between the following types of operations (and the cost associated with them) (1) local computation (C_l) (2) communication with a neighbor (C_n), and (3) communication with a arbitrary processor (C_a). In addition we distinguish between the costs of blocking caused by the use of barrier synchronizations (T_{bar}) and point-to-point communications (T_{sr}). O_{bar} is the overhead to implement barrier synchronization. The cost of communication operations immediately preceded by a barrier is C_n (or C_a), whereas when such an operation is not preceded by a barrier the cost is $C_n + T_{sr}$ (or $C_a + T_{sr}$). Note that the latter case is strictly an upper bound; as it represents the case of a thread arriving at the blocking receive T_{sr} units before a corresponding thread executes a send. We will show that even with this conservative estimate, our optimizations show an improvement.

The total time is computed by including the cost of each of these components as follows:

$$T_{ex} = n_l C_l + n_{bar}(T_{bar} + O_{bar}) + n_{bn} C_n + n_n(C_n + T_{sr}) + n_{ba} C_a + n_a(C_a + T_{sr})$$

where n_l is the number of local operations, n_{bar} the number of barriers, n_{bn} the number of neighbor communications immediately preceded by a barrier, n_n the number of neighbor communications not preceded by a barrier, n_{ba} the number of arbitary communications immediately preceded by a barrier, n_a are the number of arbitrary communications not preceded by a barrier. We assume that O_{bar} is algorithmically dependent on the number of processors, p. Specifically, $O_{bar} = c * \log p$.

An Example

The following program fragment is from an application modelling diffusion aggregation in fluid flow (further details can be found in [ABCD93]). It contains a guarded par statement block, where the guard can be evaluated locally.

```
index_set I:i = {0..N-1};
index_set J:j = {0..M-1};
int particle[N][M], s[N][M], swapped[N][M], count[N][M], tmp[N][M];
int A, P, D;
...
```

```
par (I,J) st (particle[i][j] == A)
{
    s[i][j] = P;                           /* s1 */
    s[i-1][j] = D;                         /* s2 */
    swapped[i-1][j] = 1;                   /* s3 */

    tmp[i][j] = count[i][j];               /* s4 */
    count[i][j] = count[i-1][j];           /* s5 */
    count[i-1][j] = tmp[i][j];             /* s6 */
}
```

In the absence of optimizations that reduce the number of barriers, an upper bound on the cost of implementing this fragment is given by:

$$T_{ex} = 3C_l + 4C_n + 14(T_{bar} + O_{bar})$$

By analyzing the code fragment for dependency relationships, it is possible to reduce the barrier synchronizations. The following is the transformed code fragment with the looser synchronization specified using *arb* statements:

```
arb (I,J) {
    guard[i][j] = (particle[i][j] == A);
    if (guard[i][j]) s[i][j] = P;          /* s1 */
}
arb (I,J) st (guard[i][j])
{
    s[i-1][j] = D;                         /* s2 */
    swapped[i-1][j] = 1;                   /* s3 */
    tmp[i][j] = count[i][j];               /* s4 */
    count_copy[i][j] = count[i-1][j];      /* s5 - part 1 */
}
    count[i][j] = count_copy[i][j];        /* s5 - part 2 */
arb (I,J) st (guard[i][j])
    count[i-1][j] = tmp[i][j];             /* s6 */
```

The execution time of the fragment after implementing these optimizations is given by the following equation:

$$T_{ex} = 4C_l + 3(T_{bar} + O_{bar}) + 2C_n + 2(C_n + T_{sr})$$

All subscript expressions in the program fragment can be evaluated at compile time. This allows us to replace each barrier by a *send–blocking receive* pair using the optimizations discussed in the previous secton. The final code fragment that results from this transformation is as follows:

```
arb (I,J) {
    guard[i][j] = (particle[i][j] == A);
    send(guard, (i-1,j));
```

```
        blx_recv(n_guard, (i+1,1));
        send(guard, (i+1,j));
        blx_recv(p_guard, (i-1,1));

        if (guard[i][j])      s[i][j] = P;                    /* s1 */
        if (guard[i][j])      send (D, (i-1,j));
        if (n_guard[i][j])    blx_recv (s, (i+1,j));          /* s2 */
        if (guard[i][j])      swapped[i-1][j] = 1;            /* s3 */
        if (guard[i][j])      tmp[i][j] = count[i][j];        /* s4 */
        if (p_guard[i][j])    send (count, (i+1, j));
        if (guard[i][j])      blx_recv (count, (i-1,j));      /* s5 */
        if (guard[i][j])      send (tmp, (i-1, j));
        if (n_guard[i][j])    blx_recv (count, (i+1,j));      /* s6 */
    }
```

With this optimization, our estimated execution time becomes:

$$T_{ex} = 3C_l + 6(C_n + T_{sr})$$

The execution time for the three programs may be compared using representative costs for the different operations We assume $C_l = 1$ unit, $C_n = 200$ units. [4] A normal distribution of arrival times gives T_{sr}:T_{bar} = 1:4. Hence, we assume $T_{sr} = k$ units and $T_{bar} = 4k$ units. Figure 8 shows two graphs. The first graph shows execution times for each of the three programs for increasing values of O_{bar} (while maintaining blocking times constant for $k = 2$). We have assumed that very efficient barrier implementations exist by considering the values for O_{bar} in the range $2 * C_n \leq O_{bar} \leq 6 * C_n$.[5] Even in the presence of efficient barriers the analysis indicates that the performance of the optimized program may be improved by upto 54% [6] as compared with the program which simply minimized the number of barrier synchronizations. The second graph shows that even with the help of extremely efficient barrier synchronizations ($O_{bar} = 2 * C_n$), the optimizations will yield significant performance improvements as the average block time is increases.

5 Conclusion

In this paper, we have addressed the issue of reducing synchronization costs when implementing a data-parallel language on an asynchronous architecture. The synchronization issue has been addressed from two perspectives: first, we have described language constructs that allow the programmer to specify that different parts of a data-parallel program be synchronized at different

[4] These assumptions are based on a 25MHz clock, hence 0.64 μsecs to perform an assignment, and a 128 μsecs message transmission time to a neighbor[SSS88]

[5] The lower bound of $2 * C_n$ represents the implementation where each processor sends a ready signal to a control processor which, in turn, informs all processors of arrival at the barrier. The upper bound of $6 * C_n$ assumes $p = 64$ processors and a $\log p * C_n$ overhead.

[6] Percentage improvement for $O_{bar} = 3C_n$.

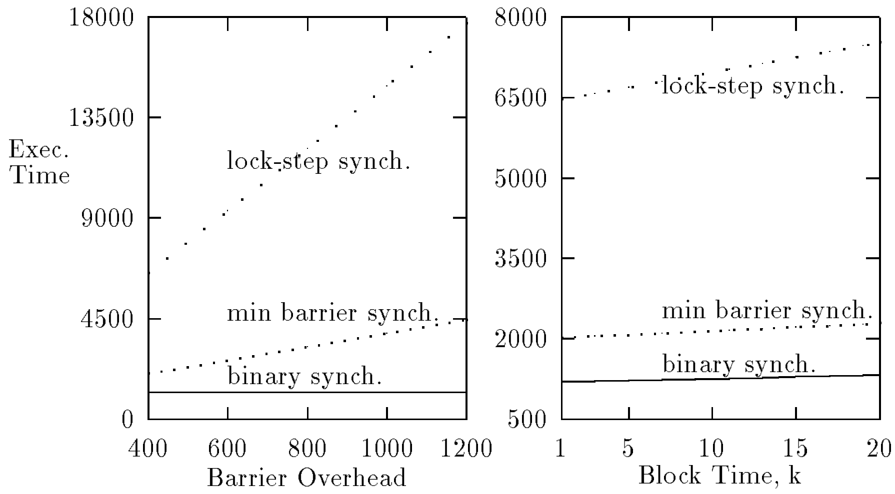

Figure 8: Performance Comparison

levels of granularity. This allows the programmer to take on the burden of explicit data management to reduce synchronization costs. It also supports an iterative approach to program design where programs are initially designed assuming operation level synchronization and may be later refined to use a weaker synchronization granularity. Secondly, we have shown how existing tools and algorithms for data dependency analysis can be used by the compiler to both reduce the number of barriers and to replace global barriers by cheaper clustered synchronizations. Although the techniques presented in the paper are general purpose, we have described them in the context of a data-parallel language called UC developed at UCLA.

References

[ABCD93] V. Austel, R. Bagrodia, M. Chandy, and M. Dhagat. Reductions + Relations = Data-Parallelism. Technical report, University of California, Los Angeles, CA 90024, April 1993.

[AK87] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Program to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–543, October 1987.

[BA92] R. Bagrodia and V. Austel. *UC User Manual.* Computer Science Department, University of California at Los Angeles, 1992.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):1–33, February 1993.

[For92] High Performance Fortran Forum. High Performance Fortran Language Specification. DRAFT, November 1992.

[FOW87] J. Ferrante, K. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3), July 1987.

[Gav72] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[Gup89] Rajiv Gupta. The fuzzy barrier: A mechanism for high-speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–64, April 1989.

[HQL+91] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Trans. on Parallel and Distributed Systems*, July 1991.

[MA87] S. K. Midkiff and Padua D. A. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[MR90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, 1990.

[PW86] D.A. Padua and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[QHS91] M. Quinn, P. Hatcher, and B. Seevers. Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor. In A. Nicolau, D. Gelernter, Gross T., and Padua D., editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 20, pages 385–401. MIT Press, Cambridge, Massachusetts, 1991.

[RS87] J.R. Rose and G.L. Steele. C*: An Extended C Language for Data Parallel Programming. PL-87.5, Thinking Machines Corporation, March 1987.

[RSW91] M. Rosing, R. B. Schnabel, and R. B. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[SSS88] C. L. Seitz, J. Seizovic, and W. K. Su. The C Programmer's Abbreviated Guide to Multicomputer Programming. Technical Report Caltech-CS-TR-88-1, Caltech, January 1988.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.