Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors

María Jesús Garzarán, Milos Prvulovic[†], Ye Zhang[†], Alin Jula[‡], Hao Yu[‡], Lawrence Rauchwerger[‡], and Josep Torrellas[†]
Universidad de Zaragoza, Spain, http://www.cps.unizar.es/deps/DIIS/gaz
[†]University of Illinois at Urbana-Champaign, http://iacoma.cs.uiuc.edu
[‡]Texas A&M University, http://www.cs.tamu.edu/faculty/rwerger

Abstract

Reductions are important and time-consuming operations in many scientific codes. Effective parallelization of reductions is a critical transformation for loop parallelization, especially for sparse, dynamic applications. Unfortunately, conventional reduction parallelization algorithms are not scalable.

In this paper, we present new architectural support that significantly speeds-up parallel reduction and makes it scalable in shared-memory multiprocessors. The required architectural changes are mostly confined to the directory controllers. Experimental results based on simulations show that the proposed support is very effective. While conventional software-only reduction parallelization delivers average speedups of only 2.7 for 16 processors, our scheme delivers average speedups of 7.6.

1 Introduction

During the last decade, programmers have obtained increasing help from parallelizing compilers. Such compilers help detect and exploit parallelism in sequential programs. They also perform other transformations to reduce or hide memory latency, which is crucial in modern parallel machines.

In scientific codes, an important class of operations that compilers have attempted to parallelize is *reduction* operations. A reduction operation occurs when an associative and commutative operator $\otimes$ operates on a variable x as in $x = x \otimes expression$, where x does not occur in $expression$ or in any other place in the loop.

Parallelization of reductions is crucial to the overall performance of many parallel codes. Transforming reductions for parallel execution requires two steps. First, data dependence or equivalent analysis is needed to prove that the operation is indeed a reduction. Second, the sequential computation of the reduction must be replaced with a parallel algorithm.

In parallel machines of medium to large size, the reduction algorithm is often replaced by a parallel prefix or recursive doubling computation [15, 21]. For reductions on array elements, a typical implementation is to have each processor accumulate partial reduction results in a private array. Then, after the loop is executed, a cross-processor merging phase combines the partial results of all the processors into the original, shared array.

Unfortunately, such an algorithm can be very inefficient in scalable shared-memory machines when the reduction array is large and sparsely accessed. Indeed, the merging phase of the algorithm induces many remote memory accesses and its work does not decrease with the number of processors. As a result, parallel reduction is slow and not scalable.

In this paper, we propose new architectural support to speed-up parallel reductions in scalable shared-memory multiprocessors. Our support eliminates the need for the costly merging phase, and effectively realizes truly-scalable parallel reduction. The proposed support consists of architectural modifications that are mostly confined to the directory controllers.

Results based on simulations show that the proposed support is very effective. While conventional software-only parallelization delivers an average speedup of 2.7 for 16 processors, the proposed scheme delivers an average speedup of 7.6.

This paper is organized as follows: Section 2 discusses the parallelization of reductions in software, Section 3 presents our new architectural support, Section 4 describes our evaluation methodology, Section 5 evaluates our proposed support, Section 6 outlines how the support can also be used for another problem, Section 7 presents related work, and Section 8 concludes.

0-7695-1363-8/01 $10.00 © 2001 IEEE

2 Parallelization of Reductions in Software

2.1 Background Concepts

A loop can be executed in parallel without synchronization only if its outcome does not depend upon the execution order of different iterations. To determine whether or not the order of iterations affects the semantics of the loop, we need to analyze the data dependences across iterations (or cross-iteration dependences) [1]. There are three types of data dependences: *flow* (read after write), *anti* (write after read), and *output* (write after write).

If there are no dependences across iterations, the loop can be executed in parallel. Such a loop is called a *doall* loop. If there are cross-iteration dependences, we must insert synchronization or eliminate the dependences before we can execute the loop in parallel.

If there are only anti or output dependences in the loop, we can eliminate them by applying *privatization.* With privatization, each processor creates a private copy of the variables that cause anti or output dependences. During the parallel execution, each processor operates on its own private copy.

Figure 1(a) shows an example of a loop that can be parallelized through privatization. There is an anti dependence between the read to variable *Temp* in line 4 and the write to *Temp* in line 2 in the next iteration. Furthermore, there is an output dependence between the write to *Temp* in line 2 in one iteration and the next one. By privatizing *Temp*, these dependences are removed and the loop can be executed in parallel.

```
1   for(i=0;i<n;i+=2){
2       Temp=a[i+1];
3       a[i+1]=a[i];
4       a[i]=Temp;
5   }
```

(a)

```
1   for(i=1;i<n;i++)
2       A[i]+=A[i-1];
```

(b)

Figure 1. Loops with anti and output dependences (a) and flow dependences (b).

If there are flow dependences across iterations, the loop cannot generally be executed in parallel. For example, the loop in Figure 1(b) has a flow dependence in line 2 between consecutive iterations. In this case, iteration i needs the value that is produced in iteration $i - 1$. As a result, the loop cannot be executed in parallel.

2.2 Parallelizing Reductions

A special and frequent case of flow dependence occurs in loops that perform reduction operations. A reduction operation occurs when an associative and commutative operator $\otimes$ operates on a variable x as in $x = x \otimes expression$, where x does not occur in $expression$ or in any other place in the loop. In such a case, x is a reduction variable.

A simplified example of reduction is shown in Figure 2. In the figure, array *w* is a reduction variable. Note that the pattern of access to a reduction variable is a read followed by a write. Therefore, there may be flow dependences across iterations. As a result, the loop cannot be run in parallel.

```
1   for(i=0;i<Nodes;i++)
2       w[x[i]]+=expression;
```

Figure 2. Loop with a reduction operation.

Parallelizing loops with reductions involves two steps: recognizing the reduction variable and transforming the loop for parallelism. Recognizing the reduction variable involves several steps [30]. First, the compiler syntactically pattern-matches the loop statements with the template of a general reduction ($x = x \otimes expression$). In our example, the statement in line 2 matches the pattern. Then, the operator ($+$ in our example) is checked to determine if it is commutative and associative. Finally, data dependence or equivalent analysis is performed to verify that the suspected reduction variable is not accessed anywhere else in the loop. In our example, all of these conditions are satisfied for *w*.

Once the reduction variable is recognized, the loop is transformed by replacing the reduction statement with an equivalent parallel algorithm. For this, there are several known methods. The two most common ones are as follows:

- Enclose the access to the reduction variable in an unordered critical section [8, 30]. Alternatively, we can access the variable with an atomic *fetch-and-op* operation. The main drawback of this method is that it is not scalable, as the contention for the critical section increases with the number of processors. Thus, it is recommended only for low-contention reductions.
- Exploit the fact that a reduction operation is an associative and commutative recurrence. Therefore, it can be parallelized using a parallel prefix or a recursive doubling algorithm [15, 21]. This approach is more scalable.

For reductions on array elements, a commonly-used implementation of the second method is to create, for each processor, a private version of the reduction array initialized with the neutral element of the reduction operator. During the execution of the parallelized loop, each processor accumulates partial results in its private array. Then, after the loop is executed, a cross-processor merging phase combines the partial results of all the processors into the shared array.

Using this approach, our example loop from Figure 2 gets transformed into the parallel loop of Figure 3. For simplicity, static scheduling is used and the code for forking and joining is omitted. Thus, we show only the code executed by each processor.

In the parallelized loop, each processor has its own array *w_priv[PID]*, where PID is the processor ID. First, each processor initializes its array with the neutral element of the reduction operator (lines 1-2). In our example, the neutral element is 0 because the reduction operator is addition. Next, each processor gets a portion of the loop and executes it, performing the reduction operation on its array *w_priv[PID]* (lines 3-4). After that, all processors synchronize (line 5). Then, they all perform a *Merging* step, where the partial results accumulated in the different *w_priv[PID]* arrays are merged into the shared array *w*.

```
// Initialize the private reduction array
1   for(i=0;i<NumCols;i++)
2       w_priv[PID][i]=0;
// The range 0..Nodes is split among the processors
3   for(i=MyNodesBegin;i<MyNodesEnd;i++)
4       w_priv[PID][x[i]]+=expression;
5   barrier();
//The range of indices of w is split among processors
6   for(i=MyColsBegin;i<MyColsEnd;i++)
7       for(p=0;p<NumProcessors;p++)
8           w[i]+=w_priv[p][i];
9   barrier();
```

Figure 3. Code resulting from parallelizing the loop in Figure 2.

In the case of scalars, this merging step can be parallelized through recursive doubling. In the case of arrays, however, it is more efficient to parallelize it by having each processor perform merging for a sub-range of the shared array. Thus, in our example each processor processes a portion of *w* element by element (line 6). For each element, the processor in charge of processing it takes the partial result of each processor (line 7) and combines it into its shared counterpart (line 8). Finally, another global synchronization is performed (line 9), to guarantee that the subsequent code accesses see only the fully merged array w.

2.3 Drawbacks in Scalable Multiprocessors

This implementation of reduction parallelization has two important drawbacks in scalable shared-memory multiprocessors with large memory access latencies: many remote misses in the merging phase and cache sweeping in the initialization and merging phases.

The merging phase necessarily suffers many remote misses. Indeed, for each shared array element that a processor accesses in line 8, all but one of the corresponding private array elements are in remote memory locations. Because of this, the merging operation (also called *merge-out*) can be very time consuming.

Note that the time needed to perform the merging does not decrease when more processors are used. With more processors, each processor has to perform combining for fewer elements of the shared array. However, each element requires more work because more partial results need to be combined. Specifically, consider an array of size s and p processors. The merging step requires that each processor combine p sub-arrays of size s/p. As a result, the total merging time is proportional to $p * s/p = s$, which does not depend on the number of processors.

The problem gets worse when the access pattern of the reduction is sparse. In this case, the merging operation performs a lot of unnecessary work, since it operates on many elements that still contain the neutral element. To improve this case, each processor could use a compact private data structure such as a hash table instead of a full private array. With this approach, however, improving the merging phase comes at the cost of slowing down the main computation phase. The reason is that addressing this compact structure requires indirection, which is more expensive than the simple addressing of array elements.

The second problem, namely cache sweeping, occurs in the initialization (lines 1-2) and merging (lines 6-8) phases. Cache sweeping in the initialization may cause additional cache misses in the main computation phase (lines 3-4). Cache sweeping in the merging phase may cause additional misses in the code that follows the reduction loop.

3 Private Cache-Line Reduction (PCLR)

To address the problems discussed in Section 2.3, we propose to add new architectural support to scalable shared-memory multiprocessors. We call the new scheme *Private Cache-Line Reduction (PCLR)*. In this section, we give an overview of the scheme and then propose an implementation.

3.1 Overview of PCLR

The essence of PCLR is that each processor participating in the reduction uses *non-coherent lines in its cache* as temporary private storage to accumulate its partial results of the reduction. Moreover, if these lines are displaced from the cache, their value is *automatically accumulated* onto the shared reduction variable in memory. Finally, since the cache lines are non-coherent, cache misses are satisfied from within the local node by returning a line *filled with* *neutral elements.* Figure 4 shows a representation of the scheme.

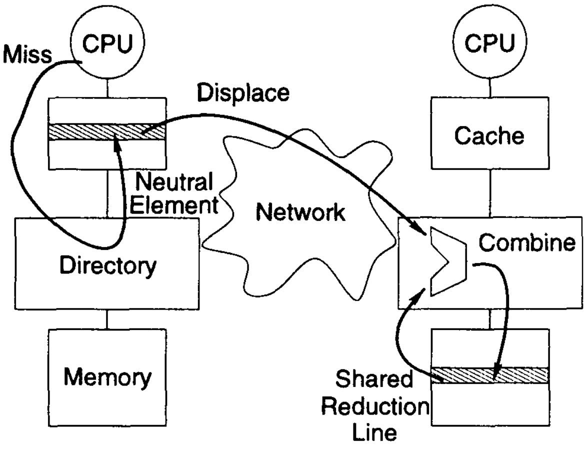

Figure 4. Representation of how PCLR works.

With this approach, the processors are relieved of the initialization and merge-out work, therefore eliminating the two problems pointed out in Section 2.3. Also, since the approach is still based on computing partial results and combining them, the reduction is performed with no critical sections.

The initialization phase is avoided by initializing the reduction lines on demand, as they are brought into the cache on cache misses. Since the cache is used as private storage to accumulate the partial results, there is no need to allocate any private array in memory. On a cache miss to a reduction line, the local directory controller intercepts the request and services it by supplying a line of neutral elements.

The merging phase is avoided by combining the reduction cache lines in the background, as they are displaced from the cache during parallel loop execution. As each displaced reduction line reaches the home of the shared reduction variable, the directory controller combines its contents with the shared reduction variable in memory. Meanwhile, the processors continue processing the loop without any interruption.

When the parallel loop ends, some partial results may still remain in the caches. They must be explicitly flushed so that they are correctly combined with the shared data before any further code is executed. This flush step takes much less time than the ordinary merging phase of Figure 3. There are two reasons for it. First, it has less combining to perform, as most of it has already been performed through displacements during the loop execution. In fact, the work to do is at worst proportional to the size of the cache, rather than to the size of the shared array. The second reason is that the processor issues no remote loads. Instead, it simply sends all the partial results to their homes, where the directory controller combines the data.

With PCLR support, the code in Figure 2 becomes the one in Figure 5. Note that we have added a call to a function that configures the machine for PCLR before the loop execution. As in Figure 3, this example is also simplified by using static scheduling and omitting the forking and joining code. In the rest of this section, we present an implementation of PCLR.

```
ConfigHardware(arguments);
// The range 0..Nodes is split among the processors
for(i=MyNodesBegin;i<MyNodesEnd;i++)
    w[x[i]]+=expression;
CacheFlush();
barrier();
```

Figure 5. Parallelized reduction code under PCLR.

3.2 Implementation of PCLR

Any implementation of PCLR has to consider the following issues: differentiation of reduction data (Sections 3.2.1 and 3.2.5), support for on-demand initialization (Section 3.2.2) and combining (Section 3.2.3) of lines, configuration of the hardware (Section 3.2.4), and atomicity guarantees (Section 3.2.6). We discuss these issues in this section.

In the following discussion, we assume a CC-NUMA architecture such as the one in Figure 4. Each node in the machine has a directory controller that snoops and potentially intervenes on all requests and write-backs issued by the local cache, even if they are directed to remote nodes.

3.2.1 Differentiating Reduction Data

While the data used in reduction operations remain in the cache, they are read and written just like regular, non-reduction data. However cache misses and displacements of reduction data require special treatment. Consequently, any implementation of PCLR has to provide a way to distinguish reduction data from regular data.

A simple way of doing so is to use special load and store instructions for "reduction" accesses. Cache lines accessed by these special instructions are marked as containing reduction data by putting them into a special "reduction" state. In this state, a processor can read and write the line without sending invalidations, even though other processors may be caching the same memory line. Misses by reduction loads and displacements of lines in the reduction state cause special transactions that are recognized by the local and home directories, respectively.

Note that we assume that reduction and regular data never share a cache line. Although it would be possible to enhance our scheme to support line sharing, alignment of reduction data on cache line boundaries is beneficial even without PCLR. Consequently, we assume that the compiler guarantees no line sharing.

In the following, we explain the rest of PCLR assuming this simple approach to differentiating reduction data. In Section 3.2.5, we propose a more advanced scheme for reduction data differentiation that allows using unmodified or slightly modified processors and caches.

3.2.2 On-Demand Initialization of Reduction Lines

When a reduction load misses in the cache, a specially-marked cache line read transaction is issued to the memory system. The local directory controller intercepts the request and satisfies it by returning a line initialized with *neutral elements* for the particular reduction operation. The line is loaded into the cache in the reduction state.

A reduction load may hit in the cache on a line that is not in the reduction state. This may occur if the line had been accessed prior to the reduction loop with plain accesses and happened to linger in the cache. In this case, if the line is in state dirty, it is written back to memory in a plain write-back. Irrespective of its state, the line is then invalidated. Finally, the cache issues a reduction read miss as indicated above.

3.2.3 On-Demand Combining of Partial Results

When a line in the reduction state is displaced from the cache, a specially-marked write-back transaction is issued to the memory system. Once the write-back arrives at its home, the directory controller reads the previous contents of the line from memory, combines it with the newly-arrived partial result, and stores the updated line back to memory. The combining of the lines is done according to the reduction operator in the code, and is performed for every single element in the line. Note that those elements of the displaced line that were not accessed by the processor still contain the neutral element, so the effect of merging them with memory content is that the memory content is unchanged.

To combine the lines, the directory controller has to be enhanced with execution units that support the required reduction operators. Since a cache line contains several individual data elements, such execution units may become a bottleneck if their performance is too low. Luckily, all the elements of a line can be processed in parallel or in a pipelined fashion. Consequently, it is not too difficult to improve the performance by pipelining these execution units or adding more units.

These execution units should include an integer ALU for integer operations. For floating-point operations, having a full floating-point unit would be more general, but would also increase the complexity of the directory controller significantly. Our experience with the applications in Section 4.2 suggests that multiplication is rarely used as a reduction operator. Thus, for floating-point operations, having a floating-point adder and comparator is sufficient.

Finally, it is possible that the reduction data had been accessed prior to the reduction loop with plain accesses, and still lingers in several caches when the reduction loop starts. To handle this case, when the home directory controller receives a write-back for the line, it always checks the list of sharer processors for the line in the directory. Note that misses due to the reduction accesses do not go to the home. Thus, the home only has sharing information about non-reduction sharers. If the line is in a (non-reduction) dirty state in a cache, the controller recalls the line and writes it back to shared memory before performing any combining. The controller also sends invalidations to all (non-reduction) sharer processors. After the first reduction write-back of a line, the list of sharers at its home is empty for the remainder of the reduction loop and causes no further invalidation or recall messages.

3.2.4 Configuring the Hardware

Before executing a reduction loop, each processor issues a system call to inform the directory controller in its node about the data type and the operation of the reduction. This is shown in line 1 of Figure 5. With this simple approach, we can only support one type of reduction operation per parallel section. In our example of Figure 5, the controller must be configured to perform double-precision floating-point addition when it receives a reduction write-back.

Any loop that performs several types of reduction operation must be distributed into multiple loops, so that each loop performs only one type of reduction operation. Fortunately, loops with multiple types of reduction operation are rare.

Finally, the operating system knows if different, time-shared processes want to use different types of reduction operations. If this is the case, the operating system flushes the reduction data from the caches when a process is preempted, and reprograms the directory controller when the process is re-scheduled.

3.2.5 Advanced Differentiation of Reduction Data

In Section 3.2.1 we explained a simple mechanism to distinguish reduction data from regular data and then explained the rest of PCLR using that simple mechanism. Now we propose a more advanced, but equivalent, mechanism that eliminates the need to modify the processor, the caches, or the coherence protocol.

In this scheme, instead of using special instructions, cache states, and protocol transactions to identify reduction data, such data are identified by using *Shadow Ad-*

dresses [5]. The scheme works as follows. In the reduction code, we use a *Shadow Array* instead of the original reduction array. For example, in Figure 5, we would use array *w_redu* instead of *w*. This shadow array is mapped to physical addresses that do not contain physical memory. However, such addresses differ from the corresponding physical addresses of the original array in a known manner. For example, they can have their most significant bit flipped. As a result, when a directory controller sees an access that addresses nonexistent memory, it will know two things. First, it will know that it is a reduction access. Second, from the physical address, it will know what location of the original array it refers to.

With this approach, we do not need to modify the hardware of the processor, caches, or coherence protocol. The only requirement is that the machine must be able to address more memory than physically installed. Then, when a directory controller sees a read miss from the local processor to nonexistent memory, it simply returns a line of neutral elements to the processor. Furthermore, when a directory controller sees the write-back of a line from the local processor to nonexistent memory, it will forward it to the home of the corresponding element of the original array. Finally, when a directory controller receives the write-back of a line from a remote processor, it translates its address to the address of the corresponding element in the original array and combines the incoming data with the data in memory.

This approach requires modest compiler and operating system support. The compiler modifies the reduction code to access a shadow array instead of the original array. It also declares the shadow array and inserts a system call to tell the operating system which array is shadow of which. The operating system has to support the mapping of pages for the shadow array. Specifically, on a page fault in the shadow array, it assigns a nonexistent physical page whose number bears the expected relation to the number assigned to the corresponding original array page. Moreover, if the latter does not exist yet, it is allocated at this time.

3.2.6 Atomicity Concerns and Solutions

In PCLR, a problem occurs if a line with reduction data is displaced from the cache between a read and the corresponding write of the reduction operation. As an example, assume that the value of a variable in the line is X. This value is read into a register and updated to $X + Y$. However, before the result register is written to the cache, the line gets displaced from the cache. In this case, the partial result X will be sent to memory and accumulated onto the shared data. Then, the cache miss will be serviced with the neutral element and the variable in the line will be updated to $X+Y$. Later, a displacement of this line will cause $X+Y$ to be accumulated onto the shared data in memory. Thus, the partial result X will be accumulated onto the shared data twice.

We can solve this problem through *recovery* or *prevention.* Recovery solutions attempt to recover the correct state of computation after the problem has already occurred. Unfortunately, the problem can generally be detected only when the store misses in the cache. In our example, the recovery would involve subtracting X from either the register involved in the miss or the shared location. However, X is unknown at the time the problem is detected, as the shared location contains the combined result of other computations and X, while the offending store has $X + Y$. Instead of attempting to checkpoint the partial results or the shared value in order to enable recovery, we choose to prevent the occurrence of the problem.

Note that reduction lines in a cache do not receive external invalidations or downgrade requests that force them to write-back. Therefore, a miss on a store to a reduction line can only occur because, between the load and the store, the local processor has brought in a second line that has displaced the one with reduction data. If we ensure that the processor does not perform any access between the load and the store to the reduction variable, the displacement problem should not happen. Unfortunately, modern processors reorder independent memory accesses like those to different words of the same line and, therefore, may induce the problem. Preventing this reordering involves putting a memory fence before the load and after the corresponding store to the reduction variable. This approach is unacceptable because it would limit the performance of PCLR on modern processors.

The approach that we use is the *pinning* of a line in the cache between a reduction load to the line and the corresponding store. We introduce two new instructions, namely *load&pin* and *store&unpin*, and add a small number of *Cache Pin Registers* (CPRs) to the processor. Each CPR has two fields: the tag field which holds the tag of the pinned line, and a *pin count* counter.

When a *load&pin* instruction is executed, a read from the cache is performed. At the same time, a CPR is allocated, its tag is set to the tag of the cache line, and the pin count is set to one. If one of the CPRs already has the tag of the line, its pin count is incremented. When a *store&unpin* instruction is executed, a store to the cache is performed. At the same time, the pin count for the matching CPR is decremented. If after this the pin count is zero, the CPR is freed. Before a displacement of a cache line is allowed, the tags of the CPRs are checked. If any of the active CPRs has a matching tag, the displacement is prevented until the line is no longer pinned in the cache.

With this support, all micro-architectural features found in modern microprocessors can still be used, including out-of-order instruction issue, speculative execution, instruction squashing, and memory renaming. However, care must be taken to keep the CPRs up-to-date. For example, if a speculatively executed *load&pin* has to be squashed, the hardware needs to decrement the corresponding pin count and possibly free the CPR. Similarly, consider memory renaming from a *store&unpin* to a *load&pin* of the same address. In this case, even though the load is transformed into a register-to-register transfer, the CPR for the *load&pin* still needs to be operated on.

Finally, if all CPRs are in use when one more is needed, or a pin counter saturates, the instruction is delayed until a CPR is free or the counter is decremented. Because CPRs are needed to allow instruction reordering by the processor, this delay cannot cause deadlocks. In fact, even if a processor has only one CPR, it can correctly execute any code. With more CPRs, the compiler can be more aggressive about instruction scheduling. In practice, we have found that a small number of CPRs (8) is sufficient to maintain good performance.

3.3 Summary

The PCLR scheme addresses the problems of parallel reductions in scalable shared-memory multiprocessors as discussed in Section 2.3. PCLR has two main advantages. First, it uses cache lines as the only private storage and initializes them on demand. As a result, there is no need to allocate private data structures or to perform a cache-sweeping initialization loop. Second, it performs the combining of the partial results with their shared counterparts on demand, as the reduction loop executes. As a result, there is no need for a costly merging step that involves sweeping the cache and many remote misses. All that is needed is to flush the reduction data from the caches at the end of the loop. These two advantages are particularly important when the reduction access patterns are sparse.

Most PCLR modifications are in the directory controllers, which perform special actions on read misses and write backs. With the use of shadow addresses, the only modification to the processor and caches is the ability to pin and unpin lines in the caches through the *load&pin* and *store&unpin* instructions. It can be argued that these instructions could also be useful for other functions in modern processors.

4 Evaluation Methodology

We evaluate the PCLR scheme using simulations driven by several applications. In this section, we describe the simulation environment and the applications.

4.1 Simulation Environment

We use an execution-driven simulation environment based on an extension to MINT [26] that includes a dynamic superscalar processor model [14]. The architecture modeled is a CC-NUMA multiprocessor with up to 16 nodes. Each node contains a fraction of the shared memory and the directory, as well as a processor and a two-level cache hierarchy with a write-back policy. The processor is a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. Table 1 lists the main characteristics of the architecture. Contention is accurately modeled in the entire system, except in the network, where it is modeled only at the source and destination ports.

Processor Parameters	Memory Parameters
4-issue dynamic, 1 GHz	L1, L2 size: 32 KB, 512 KB
Int, fp, ld/st FU: 4, 2, 2	L1, L2 assoc: 2 way, 4 way
Inst. window: 64	L1, L2 size: 64 B, 64 B
Pending ld, st: 8, 16	L1, L2 latency: 2, 10 cycles
Branch penalty: 4 cycles	Local memory latency: 104 cycles
Int, fp rename regs: 64, 64	2-hop memory latency: 297 cycles

Table 1. Architectural characteristics of the modeled CC-NUMA. The latencies shown measure contention-free round trips from the processor in processor cycles.

The system uses a directory-based cache coherence protocol along the lines of DASH [22]. Each directory controller has been enhanced with a single double-precision floating-point add unit. Both the directory controller and the floating point-unit are clocked at 1/3 of the processor's frequency. The floating-point unit is fully pipelined, so it can start a new addition every three processor cycles. Its latency is 2 cycles (6 processor cycles). Floating-point addition is the only reduction operation that appears in our applications (Section 4.2).

Private data are allocated locally. Pages of shared data are allocated in the memory module of the first processor that accesses them. Our experiments show that this allocation policy for shared data achieves the best performance results for both the baseline and the PCLR system.

4.2 Applications

To evaluate the PCLR system, we use a set of FORTRAN and C scientific codes. Two of them are applications: *Euler* from HPF-2 [7] and *Equake* from SPECfp2000 [13]. The three other codes are kernels: *Vml* from Sparse BLAS [6], *Charmm* from [4], and *Nbf* from the GROMOS molecular dynamics benchmark [11].

All of these codes have loops with reduction operations. Table 2 lists the loops that we simulate in each application and their weight relative to the total *sequential* execution

Appl.	Names of Loops	% of Tseq	# of Invocations	Iters. per Invocation	Instruc. per Iter.	Red. Ops. per Iter.	Red. Array Size (KB)	Lines Flushed	Lines Displaced
Euler	*dflux_do[100,200]* *psmoo_do20* *eflux_do[100,200,300]*	84.7	120	59863	118	14	686.6	3261	2117
Equake	*smvp*	50.0	3855	30169	550	22	707.1	742	580
Vml	*VecMult_CAB*	89.4	1	4929	135	6	40.0	168	0
Charmm	*dynamc_do*	82.8	1	82944	420	54	1947.0	1849	330
Nbf	*nbf_do50*	99.1	1	128000	1880	200	1000.0	238	1774
Average		81.2	795	61181	620	59	871.0	1251	960

Table 2. Application characteristics. In *Euler*, we only simulate dflux_do100, and all the numbers except Tseq correspond to this loop. The data in the last two columns of this table correspond to a single loop, and are collected through simulation of a 16-processor system.

time of the application (%Tseq). This value is obtained by profiling the applications on a single-processor Sun Ultra 5 workstation. The table also shows the number of loop invocations during program execution, the average number of iterations per invocation, the average number of instructions per iteration, the average dynamic number of reduction operations per iteration, and the size of the reduction array. The last two columns of the table will be discussed in the next section.

The loops in Table 2 are analyzed by the Polaris parallelizing compiler [2] or by hand to identify the reduction statements. Then, we modify the code to implement the parallel reduction code for the software and PCLR algorithms, as shown in Sections 2.2 and 3, respectively. For PCLR, reduction accesses are also marked with special load and store instructions to trigger special PCLR operations (Section 3.2.1) in our simulator.

In the next section we report data, including speedups, for only the sections of code described in Table 2. Also, since there is a significant variation in speedup figures across applications, we report average results using the *harmonic mean.*

5 Evaluation

5.1 Impact of PCLR

We evaluate two different implementations of our PCLR scheme. The first one is an implementation where the directory controller is hardwired. The second implementation utilizes a programmable directory controller, similar to the MAGIC micro-controller in the FLASH multiprocessor [19]. A programmable controller can provide the functionality required by PCLR without requiring hardware changes. These two implementations of PCLR are compared against a baseline system, which uses a software-only approach to parallelize reductions. The software-only approach utilizes an algorithm that accumulates partial results in private arrays and merges the data out when the loop is done, as described in Section 2.2.

Figure 6 compares the execution time of these three systems. The baseline software-only system is *Sw*. The PCLR implementation with a hardwired directory controller is *Hw*, and the implementation with a flexible programmable directory controller is *Flex*. The simulated system is a 16-node multiprocessor. For each application, the bars are normalized to *Sw*, and broken down into time spent in the initialization phase of the *Sw* scheme (*Init*), loop body execution (*Loop*), and time spent merging the partial results at the end of the loop in *Sw* or flushing the caches in *Hw* and *Flex* (*Merge*). The numbers above each bar show the speedup relative to the sequential execution of the code. In the sequential execution, all data were placed on the local memory of the single active processor.

The performance of *Hw* and *Flex* improves significantly over *Sw*. This improvement is mainly due to the elimination of the final cross-processor merging step that is required in the software-only implementation. In *Sw*, the work in this merging step is proportional to the size of the reduction array and does not decrease when more processors are available. When this time is significant relative to the time spent in the execution of the main loop, the benefits of PCLR become substantial. For example, in *Charmm* the main parallel loop alone executes with *Sw* 9 times faster than the sequential loop. However, the merging step is responsible for the poor final speedup of *Sw* (1.9 on 16 processors). As mentioned in Section 3, a second benefit of PCLR is that the initialization phase is removed. In general, this phase accounts for a relatively small fraction of the *Sw* execution time.

The *Hw* and *Flex* systems always spend less time in *Merge* than the *Sw* system. In PCLR, *Merge* only accounts for the time spent flushing the caches after the last processor has finished the execution of the parallel loop. When using PCLR, processors do not have to synchronize after the parallel loop. They can start flushing their caches as soon as

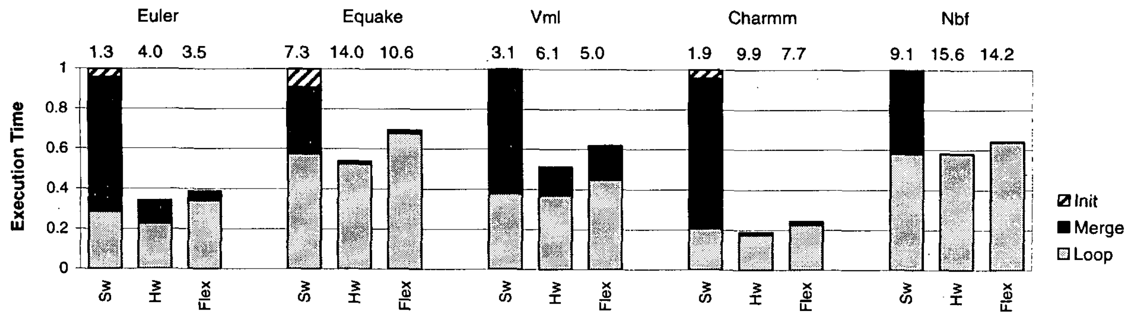

Figure 6. Execution time under different schemes for a 16-node multiprocessor. The numbers above the bars are speedups relative to the sequential execution.

they finish, and overlap this flush with the execution of the loop on other processors.

In our experiments, we do not assume special support to flush only the reduction data. Thus, the L2 cache is traversed and *all* the dirty lines (reduction or not) are written back to memory. In Table 2, the column *Lines Flushed* shows the average number of cache lines flushed by each processor. Most of these lines contain reduction data. The column *Lines Displaced* shows the average number of lines with reduction data displaced by each processor during the execution of the main reduction loop.

The differences between the *Hw* and the *Flex* schemes are mainly due to two reasons. First, with a software directory controller, all the transactions in the node have to go through the node controller, increasing the contention. Second, the software directory controller takes longer to process individual transactions. To accurately simulate these two effects, in our simulations of *Flex* we have used the cycle counts for response time and occupancy reported for the FLASH directory controller [12]. For example, a clean read miss is serviced in 11 cycles of the directory controller. Since we assume that the directory controller is clocked at 1/3 of the processor's frequency, this corresponds to 33 cycles of the main processor. The directory controller is occupied during that time.

The figure shows that the speedups in *Flex* are, on the average, only 16% lower than in *Hw* and 136% higher than in *Sw*. Therefore, implementing PCLR using a programmable directory controller is a good trade-off.

Overall, for a 16-node multiprocessor, the *Hw* PCLR scheme achieves an average speedup of 7.6, while the software-only system delivers an average speedup of only 2.7. If PCLR is implemented with a programmable directory controller the average speedup is 6.4.

5.2 Scalability of PCLR

To evaluate the scalability of PCLR, we have simulated a multiprocessor system with 4, 8, and 16 processors. Figure 7 shows the harmonic mean of the speedups delivered by the different mechanisms. It can be seen that PCLR (both *Hw* and *Flex*) scale well. However, the *Sw* scheme scales poorly. As explained in Section 2.2, the time of the merging step in *Sw* does not decrease when more processors are available. If the main loop scales well, the merging step limits the achievable speedups according to Amdahl's law.

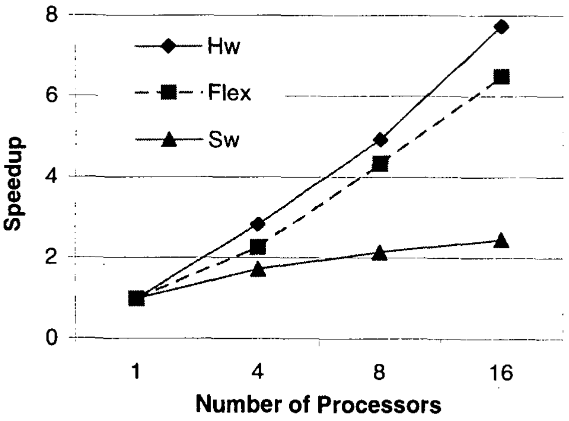

Figure 7. speedups delivered by the different mechanisms (harmonic mean).

5.3 Impact of FP-Unit Speed

In previous sections we have assumed that the floating-point unit in the directory controller was clocked at 1/3 of the processor's frequency. To determine whether this unit is a point of contention in our PCLR system, we evaluate a system with a faster unit. Thus, Figure 8 compares the previous system (*Hw*) with a system where the floating-point unit in the directory is clocked at the full frequency of the processor (*Fast*). We can see from the Figure that, although the execution times of some applications improve, the improvements are not significant. Therefore, even the relatively slow floating-point unit is not a bottleneck in our system.

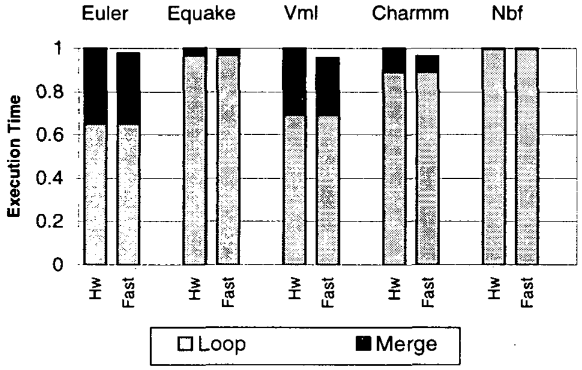

Figure 8. Comparing the performance with floating-point units of different frequencies.

6 Additional Use of PCLR

The PCLR scheme can also be used to speed-up another algorithm, namely the dynamic last value assignment. In this section, we explain the dynamic last value assignment problem (Section 6.1) and show how PCLR can be used (Section 6.2).

6.1 Dynamic Last Value Assignment in Software

As explained in Section 2, privatization is a common technique used to parallelize loops with anti and output dependences. When privatization is used, if the value of the privatized variable is needed after the parallel loop, a *last value assignment* has to be performed. Specifically, after the loop execution is complete, the shared counterpart of the privatized variable has to be updated with the value produced by the highest writing iteration. If the compiler can determine which iteration is that, it generates the code that puts this value in the shared variable. In the common case when all iterations write to the privatized variable, the last writing iteration is the last iteration of the loop. In this case, the compiler can simply peel off this last iteration and make it write directly to the shared variable.

However, when the last writing iteration cannot be determined at compile time, *dynamic last value assignment* has to be performed. In this case, the main parallel loop is followed by a *copy-out* phase. This phase identifies, for each element of the shared variable, the processor that ran the highest iteration that wrote to that element.

Figure 9 shows an example of a loop that is parallelized through the privatization of array *A*. It also needs dynamic last value assignment if the elements of array *A* are read after the loop, or the compiler cannot prove they are not read.

```
for(i=0;i<N;i++)
    if(f[i]){
        A[g[i]]= . . .;
        . . .=A[g[i]];
    }
```

Figure 9. Loop to be parallelized with privatization and dynamic last value assignment.

Figure 10 shows the parallel version of the loop, with the copy-out phase. The figure assumes that array *A* contains *NumElement* elements. As usual, static scheduling is used for simplicity. As can be seen, together with the private array *A_priv[PID]*, each processor also has a private time-stamp array *A_ts[PID]*. Whenever a processor writes to an element of the private array, it also updates the corresponding element in the private time-stamp array with the number of the iteration it is executing. When the loop is done, the copy-out phase compares the private time-stamps of all the processors. Each element of the shared array is updated with the private copy of the processor where the maximum time-stamp is found.

Note that the private time-stamps *A_ts[PID]* have to be initialized to a number that is smaller that any possible iteration number (-1 in our example). Also, note that when static scheduling is used, as in our example, the processor ID can be used to update the time-stamps (*A_ts[PID][g[i]]* in line 6) instead of the iteration number, assuming that scheduling is done so that processors with increasing PIDs get iteration ranges with increasing indices.

6.2 Using PCLR for Last Value Assignment

Note that Figure 10 uses the private time-stamp arrays as reduction data, where the reduction operation is *maximum*. During the execution of the main loop, the updates to the private time-stamps compute the partial results, while the search for the maximum during the copy-out phase cor-

```
// Initialize the private time-stamp array
    for(i=0;i<NumElement;i++)
        A_ts[PID][i]=-1;
// The range 0..N is split among the processors
    for(i=MyNBegin;i<MyNEnd;i++)
        if(f[i]){
            A_priv[PID][g[i]]= . . .;
            A_ts[PID][g[i]]=PID;
            . . .=A_priv[PID][g[i]];
        }
    barrier();
// Copy-out. Range 0..NumElement is split among procs
    for(i=MyNumElemBegin;i<MyNumElemEnd;i++){
        x=-1;
        for(p=0;p<NumProcessors;p++)
            x=max(x,A_ts[p][i]);
        if(x>-1)
            A[i]=A_priv[x][i];
    }
    barrier();
```

Figure 10. Code resulting from parallelizing the loop in Figure 9 with dynamic last value assignment.

responds to the merge-out. Thus, the PCLR mechanism as explained in Section 3, can be used to speed-up the dynamic last value assignment operation. Since PCLR performs reductions without declaring private reduction arrays, we only need to declare one shared time-stamp array. Its elements have to be initialized to a number lower than any possible iteration or processor ID.

During the execution of the main loop, the time-stamps are updated using the *maximum* operator. After the parallel loop is finished and the caches are flushed, each element of the shared time-stamp array will contain the maximum time-stamp for that element. Each of these maxima will be used to identify the correct private version to copy into the corresponding element of the shared array. Thus, using PCLR speeds-up dynamic last value assignment and makes it scalable with the number of processors.

6.3 Advanced Support

A further speedup could be obtained if the final *copy-out* phase was eliminated. To do so, we could extend PCLR with additional support. Currently, PCLR speeds-up the computation of the maximum time-stamp for each data element. However, we still have to explicitly perform the copy-out.

With advanced support, caches and directory controllers could help eliminate the copy-out as follows. Every time that a line from the privatized array is displaced from the cache, the cache could also force the displacement of the corresponding time-stamps. When the displaced privatized line and time-stamps arrived at the home directory controller, a comparison would take place. An element in the privatized line would update the shared data in memory only if its time-stamp was higher than the one in shared-memory. At the end of the loop execution, a cache flush step would flush the remaining private array lines and time-stamps. Again, the home directory controller would only conditionally accept the incoming private data based on a time-stamp comparison. Overall, with this support, at the end of execution, the shared array would have the last values and no copy-out would have been necessary.

7 Related Work

Nearly all of the past work on reduction parallelization has been based on software-only transformations [8, 27]. The most related architectural work that we are aware of is the work of Larus *et al.* [20], Zhang *et al.* [28], and the work on advanced synchronization mechanisms [3, 9, 10, 16, 17, 18, 23, 24, 25, 29].

Larus *et al.* briefly mention an idea similar to PCLR as one application of their Reconcilable Shared Memory (RSM) [20]. RSM is a family of memory systems whose behavior can be controlled by the compiler. They use RSM to support programming language constructs. The paper only mentions the applicability to reduction very briefly and provides no evaluation.

Zhang *et al.* propose a modified shared-memory architecture that combines both speculative parallelization and reduction optimization [28]. In contrast to that work, which relies on a significantly modified multiprocessor architecture, we have presented relatively simple architectural support to optimize reduction parallelization. In addition, unlike in [28], our scheme assumes that the compiler has already proved that our transformation is legal.

Finally, the combining support that we propose for the directory controller is related to the existing body of work on hardware support for synchronization. Such work includes the Full/Empty bit of the HEP multiprocessor [25], the atomic Fetch&Add primitive of the NYU Ultracomputer [10], the Fetch&Op synchronization primitives of the IBM RP3 [3, 23], support for combining trees [16, 24], the memory-based synchronization primitives in Cedar [17, 18, 29], and the set of synchronization primitives proposed by Goodman *et al* [9].

8 Summary

In this paper, we have proposed new architectural support to speed-up parallel reductions in scalable shared-memory multiprocessors. The support consists of architectural modifications that are mostly confined to the directory controllers. With this support, we eliminate the final merging phase that typically appears in conventional algorithms for parallel reduction. This phase takes time that is proportional to the data size in the dense case, or to the data structure dimensions in the sparse case. With our support, parallel reduction only needs a final cache flush step that takes time proportional only to the cache size. Overall, our scheme realizes truly scalable parallel reduction. While conventional software-only parallelization delivers average speedups of 2.7 for 16 processors, the proposed scheme delivers average speedups of 7.6.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-9734471, ACI-9872126, EIA-9975018, CCR-9970488, EIA-0081307, and EIA-0072102; CICYT of Spain under grant TIC98-0511-C02; DOE under ASCI ASAP Level 2 grant B347886; Sandia National Laboratory; and by gifts from IBM, Intel, and Hewlett-Packard.

References

[1] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, 1988.

[2] W. Blume et al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[3] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proc. 1985 Intl. Conf. on Parallel Processing*, pages 782–789, 1985.

[4] B. R. Brooks et al. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, (4):187–217, 1983.

[5] J. Carter et al. Impulse: Building a Smarter Memory Controller. In *Proc. 5th Intl. Symp. on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

[6] I. Duff, M. Marrone, G. Radiacti, and C. Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for Sparse Matrices. Tech. Rep. RAL-TR-95-049, Rutherford Appleton Laboratory, 1995.

[7] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Tech. Rep. CRPC-TR94492, Rice University, Nov. 1994.

[8] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, Aug. 1991.

[9] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. 3rd Intl. Conf. on Architectural Support for Prog. Lang. and OS*, pages 64–73, Apr. 1989.

[10] A. Gottlieb et al. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. In *IEEE Trans. Computers*, pages 175–189, Feb. 1983.

[11] W. Gunsteren and H. Berendsen. GROMOS: GROningen MOlecular Simulation software. Tech. Rep., Laboratory of Physical Chemistry, University of Groningen, 1988.

[12] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. 6th Intl. Conf. on Architectural Support for Prog. Lang. and Operationg Systems*, pages 274–285, Oct. 1994.

[13] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[14] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[15] C. Kruskal. Efficient Parallel Algorithms for Graph Problems. In *Proc. 1986 Intl. Conf. on Parallel Processing*, pages 869–876, Aug. 1986.

[16] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization on Multiprocessors with Shared Memory. *ACM Trans. Prog. Lang. and Systems*, 10(4):579–601, Oct. 1988.

[17] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. *Experimental Parallel Computing Architectures: Volume 1 - Special Topics in Supercomputing*, Chapter Parallel Supercomputing Today and the Cedar Approach, pages 1–23. J. J. Dongarra editor, North-Holland, New York, 1987.

[18] D. J. Kuck et al. The Cedar System and an Initial Performance Study. In *Proc. 20th Annual Intl. Symp. on Computer Architecture*, pages 213–224, May 1993.

[19] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. 21st Annual Intl. Symp. on Computer Architecture*, pages 302–313, Apr. 1994.

[20] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. 6th Intl. Conf. on Architectural Support for Prog. Lang. and OS*, pages 208–218, Oct. 1994.

[21] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[22] D. Lenoski et al. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, Mar. 1992.

[23] G. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc. 1985 Intl. Conf. on Parallel Processing*, pages 764–771, 1985.

[24] G. Pfister and A. Norton. 'Hot Spot' Contention and Combining in Multistage Interconnection Networks. In *Proc. 1985 Intl. Conf. on Parallel Processing*, pages 790–797, Aug. 1985.

[25] B. J. Smith. Architecture and Aplications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[26] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. 2nd Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, Jan. 1994.

[27] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization. In *Proc. 14th ACM Intl. Conf. on Supercomputing*, May 2000.

[28] Y. Zhang et al. A Framework for Speculative Parallelization in Distributed Shared-Memory Multiprocessors. Tech. Rep. CSRD-1582, University of Illinois at Urbana-Champaign, July 1999.

[29] C. Q. Zhu and P. C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. In *IEEE Trans. Software Engineering*, pages 726–739, June 1987.

[30] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.