International Journal of Parallel Programming, Vol. 29, No. 6, December 2001 (© 2001)

Two-Phase Barrier: A Synchronization Primitive for Improving the Processor Utilization

Inbum Jung,[1] Jongwoong Hyun, Joonwon Lee, and Joongsoo Ma

Received February 28, 2000; revised February 1, 2001

Barrier is widely used for synchronization in parallel programs. Since the process arrived earlier than others should wait at the barrier, the total processor utilization decreases. In this paper, to find the sources of the barrier waiting time, parallel programs are executed on the various grain sizes through execution-driven simulations. In simulation studies, we found that even if approximately equal amounts of work are distributed to each processor, all processes may not arrive at a barrier at the same time. The reasons are that the different numbers of cache misses and instructions within in partitioned grains result in the difference in arrival time of processors at the barrier. In this paper, the two-phased barrier is considered to reduce the blind waiting time in the traditional barrier scheme, which can be simply constructed by dividing one specific stage for the synchronization into two stages. On each stage, processes decide their stall or not, which is dependent on the current execution state of grains running on any given processors. Simulation results show that the reduced barrier waiting times attributed to the two-phased barrier contribute to the performance improvement of parallel programs.

KEY WORDS: Barrier; synchronization; cache miss; grain size; shared memory multiprocessors.

[1] Corresponding author: Division of Computer, Information & Telecommunication Engineering, Kangwon National University, 192-1, Hyoja 2-Dong Chunchon, Kangwon-Do 200-701, Korea. E-mail: ibjung@kangwon.ac.kr

0885-7458/01/1200-0607/0 © 2001 Plenum Publishing Corporation

1. INTRODUCTION

Barriers are commonly used for synchronization among all the processors in parallel programs. Upon reaching a barrier, the processor must wait until all processors reach the barrier. Since processors that are blocked at the barrier are essentially idling, they cannot contribute to any useful work. Barriers may be automatically inserted by a parallelizing compiler or may be introduced explicitly by the programmer. Even if the compiler or programmer distributes the computation so that all processors execute an identical code, they may not arrive at a barrier at the same time. If the code contains conditional statements, different processors may follow different control paths, and thus they executes varying number of instructions. Furthermore, the times for memory accesses may vary for different processors. The processors suffering from cache misses may fall behind in execution, hence late arrival at the barrier even if all processors are executing identical codes. In data parallel processing, the workload comprising large data sets is partitioned according to the chosen grain size. The resulting grains mean the sets of data elements and occupy the contiguous blocks of memory. Under such conditions, even if the same number of grains is allocated to each processor, the grains loaded into processors' cache result in the different cache conflict misses in each processor. In this paper, to study the sources of the barrier waiting time mentioned above, data parallel programs are executed under various grain sizes and are analyzed to identify the sources of variation in the barrier waiting time.

The waiting time at a barrier also results from poor functionality in traditional barriers, since processors reaching the barriers cannot found the execution state of the grains running on the other processors. However, for some parallel programs, if the information of partially executed grains is available in the location of barriers, the processors reaching at a barrier have a chance to execute the next iteration instead of blind waiting at the barriers. In this paper, we suggest a synchronization primitive called *two-phase barrier*, which is composed of the first-phase that notifies the current executing position within in the grains running on each processor, and the second-phase that determines whether or not to stall based on the counter value of the previous stage. The two-phase barrier reduces the barrier waiting time, since if all processors already pass the first-phase, early arriving processors at the second-phase then do not wait and can proceed to the next parallel iteration. Also, it is simply constructed by adding two counter variables in the traditional barrier structure. Our simulation results show that the proposed two-phase barrier reduces the barrier waiting time, and also improves the performance of our benchmark parallel programs.

The paper is structured as follows. Section 2 describes benchmark parallel programs and their grain size and scheduling policies of grains used in this study. In this environment, we measure the breakdown items of the processor execution time, including the barrier waiting time, under the various grain sizes and analyze the sources of the barrier waiting time. Section 3 suggests the two-phase barrier to reduce the barrier waiting time and evaluates its performance on our benchmark programs. In Section 4 related work is presented. Finally, the conclusion is presented in Section 5.

2. BENCHMARK PROGRAMS AND THEIR PERFORMANCES

2.1. Data Parallel Programs

Though parallelism can be found in various forms, data parallelism is the most intuitive and commonplace because target programs of parallel processing usually comprise large data set. A grain size determines the basic program segment chosen for parallel processing. In data parallel processing, since each data element is subject to the identical processing, the grain sizes for parallel processing are determined by dividing the total data elements by the number of available processors. In this paper, benchmark parallel programs are run under several grain sizes to observe how the barrier waiting time is affected by the flow of control paths and by the cache misses in partitioned grains.

The partitioned grains are allocated to each processor using the static or dynamic scheduling policy.[11, 3, 16] The dynamic scheduling policy performs scheduling activities to the grains at runtime. As soon as a process completes the computation of a grain, the process begins executing the next grain in the grain queue. The dynamic scheduling improves the processor utilization because a program uses the available processors released by other processes during its execution. However, it results in both scheduling overheads to handle the grain queue and the loss of cache locality. On the other hand, the static scheduling policy designates grains to each processor before a program is executed. Since the grains allocated to each processor are not changed until the program finishes, the cache locality of programs can be fully exploited. We use the static scheduling policy in this study, since cache locality is important for achieving good performance in data parallel programs.[5, 16]

The two data parallel programs for this study are FFT (Fast Fourier Transform) and LU (LU Decomposition).[7] These programs show data parallelism, since they perform identical operations on all data elements, and thus the choice of grain sizes is clearly achieved by dividing the data elements by the multiple number of processors. In particular, since these

programs do not generate new data elements dynamically during the execution, they are easy to evaluate the performance variations according to the chosen grain sizes. All these programs are written in C and use the synchronization and sharing primitives provided by the SGI's parallel macros package. During parallel computing, a centralized barrier is used to synchronize between processors. We assume that these benchmark programs are run with eight processes on eight processors. The following describes the primary data structures, the computational behavior and the grain sizes of two data parallel programs chosen.

2.1.1. Fast Fourier Transform (FFT)

The FFT program we have used here is a classic iterative Cooley–Tukey algorithm for an n point; one-dimensional, unordered and radix-2 FFT. This program performs $\log n$ iterations of the most outer loop. Each iteration does n complex multiplications and additions. Primary data structures are two one-dimensional arrays composed of both a source point array and a result point array. The Fig. 1 is the FFT program pseudo code.

The outer loop starting at line 6 is executed $\log n$ times for an n point FFT. In every iteration of the outer loop, the array R is updated using the elements that were stored in the array S. The *chunk_size* represented in the line 5 is a unit of work executed once by all processors. Since we use the

```
1:  Procedure FFT(R, S, n)  {
2:      r = log n;
3:      m_fork( P) ;      /* set multiple processes */

4:      id = processor's label;
5:      chunk_size = grain_size × N; /* N is the number of processors used */
6:      for( m = 0; m < r –1 ; m++) { /* outer loop   */
7:          for( i = 0 ; i < n – 1 ; i++) {
8:              S[i] = R[i] ; /* address exchanging for interchanging their roles */
9:          }
10:         i = id × grain_size;
11:         for( ; i < n – 1 ; i +=chunk_size) {
12:             for( ; grain_size; grain_size --) {
13:             /* Let(b0b1 ..... br-1) be the binary representation of   i */
14:                     j = (b0 . . . bm-1 0bm+1...br-1 );
15:                     k = (b0 . . . bm-1 1bm+1...br-1);
16:                     R[i] = S[j] + S[k] × ω (b0 . . . bm-1 0bm+1...br-1) ;
17:             }
18:         }
19:         barrier(semaphore, N);
20:     }
21: }
```

Fig. 1. Fast fourier transform program using the Cooley–Turkey algorithm.

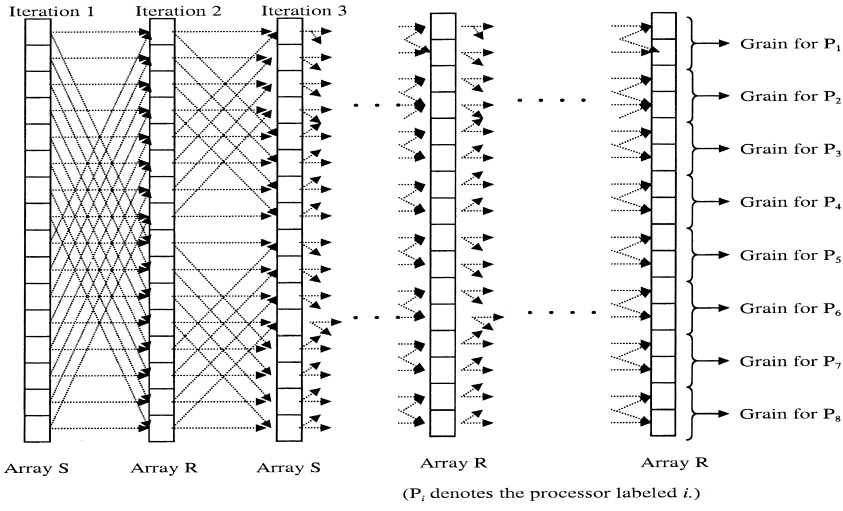

Fig. 2. A snapshot of the FFT program on parallel processing.

static scheduling policy for the scheduling of grains onto processors, each processor executes its grain within a chunk and moves to its grain of the next chunk. In the line 16, all processors update $R[i]$ by using $S[j]$ and $S[k]$ within their grains, and also compute the powers of w known as *twiddle factors*. The FFT program shows that the amount of computation of these twiddle factors is dependent on the relative position of each grain within the array R. This characteristic may cause the load imbalance even if the same number of grains is distributed to each processor. In the line 19, the traditional centralized barrier is applied to synchronize processors in the outermost loop.

For our experiments, we execute FFT on 65536 input points (1 Mbytes in size). The coarsest grain size is 8092 points, which is achieved by dividing a source point array S by eight processors. Other grain sizes considered are 4096, 2048, 1024, 512, 128, 64, 32, 16, 8, 4, and 2 points. All grain sizes balance the loads among all processors, since the same number of grains is allocated to each processor. Figure 2 shows an example of the partitioned grains under the coarsest grain size and iteration steps during the execution. As shown in this figure, arrays S and R are in turn used as a source point array or a result point array and the memory access pattern of this program follows divide-and-conquer characteristics.

2.1.2. LU Decomposition (LU)

This program decomposes matrix A as the product of a lower-triangular matrix L and an upper-triangular matrix U so that $A = L \times U$. The

```
1:  Procedure LU(A, n)  {
2:      m_fork( P ) ;       /* set multiple processes */

3:      id = processor's label;
4:      for( k = 0; k < n –1 ; k++) { /* outer loop   */
5:          for( j = k+1; j < n – 1 ; j++) {
6:              lock(lock_var);   /* spin lock */
7:              A[k, j]   = A[k, j] / A[k, k];     /* compute a pivot row */
8:              unlock(lock_var);       /* spin unlock */
9:          }
10:         /* N is the number of processors used */
11:         grain_size = remaining rows / (N × grain_divisor );
12:         chunk_size = grain_size × N ;
13:         i = (k + 1) + (id × grain_size);
14:         for( ; i < n – 1; i += chunk_size) {
15:             for( ; grain_size ; grain_size -- )
16:                 for(j = k + 1; j < n – 1 ; j++)
17:                     A[i, j ] = A[i, j] – A[i, k] × A[k, j];
18:         }
19:         barrier(semaphore, N);
20:     }
21: }
```

Fig. 3. LU program.

Fig. 3 shows the LU program pseudo code. The main data structure is a two-dimensional matrix A being decomposed. For k varying from 0 to $n-1$, this program systematically eliminates the values of the row k from those of the rows $k+1$ to $n-1$ so that the matrix of coefficients becomes upper-triangular. The pivot row's computation executes on the line 7. We regard this line as a serial code due to a relatively small amount of its computation. After a pivot row's computation, each processor uses the pivot row to modify all rows owned by it to the down of the pivot.

As computation proceeds in LU, the pivot row moves to the down and the number of rows that remain to its down decreases. As a result, the amount of data accessed and work done per the pivot row decreases. Under the pure static grain scheduling policy, the gradual decrease in workloads appears to poor processor utilization, since the processors participating in parallel computing are not used as the computation proceeds. To utilize all processors until the program completes, we recalculate the grain size by dividing the remaining rows underneath the pivot-row by both the number of processors and a *grain divisor* defined as a 32-bit integer variable. Thus, the *chunk_size* on the line 12 is determined by multiplying the number of processors by the recalculated grain size. After reconstructing the chunks, each processor executes its grain within a chunk and moves to its grain of the next chunk. These procedures are shown between the line 11 and 18. In the line 19, a traditional centralized barrier is used to synchronize processors in the outermost loop.

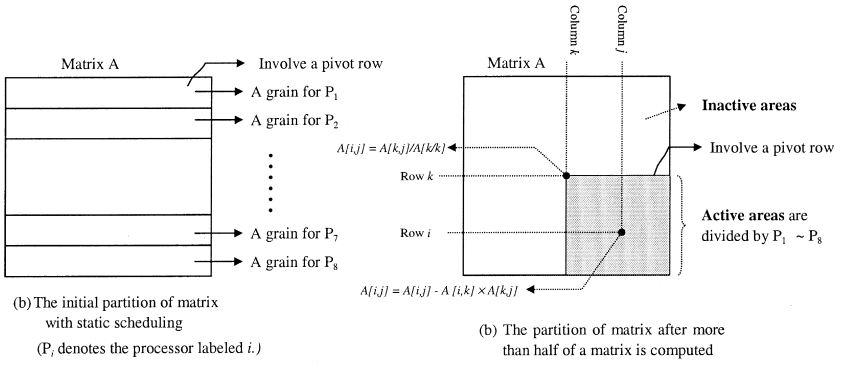

Fig. 4. A snapshot of the LU program on parallel processing.

Under our modified static grain scheduling policy, Fig. 4 shows the snapshot of the LU program during the parallel processing. Figure 4(a) shows the initial partition of a matrix A allocated to eight processors before starting the computation. Figure 4(b) shows that after more than half of a matrix is computed, only the remaining rows underneath the pivot row k are active and they are divided into eight processors. At this stage, only the lower-right $k \times k$ submatrix of A is computationally active. As shown in these figures, the number of rows allocated into each processor is decreased as the remaining active area shrinks.

Since the number of processors is fixed in our study, the grain divisor value represents the degree of granularity used in this program. The grain divisors considered are 1, 2, 3, 4, 5, 6, 7, 8, and 9. Larger grain divisors mean that finer grain sizes are applied at the remaining rows underneath the pivot-row. Thus, the finest grain size is grain divisor 9, and the coarsest grain size is grain divisor 1. For our experiments we run the LU program with a 256×256 matrix (512 Kbytes in size).

2.2. Simulation Environment

We assume the shared-memory multiprocessors system with a shared bus as a machine chosen for this study. This system is widely used and commercialized for computing servers due to its low-cost high computing power and ease of use. These machines are also called UMA (Uniform Memory Access) machines, since access to a memory location via the bus takes the same amount of time regardless of which processor is performing the access and what memory location is being accessed. Cache coherency is

Table I. Timing Parameters

Events (operations)	Penalties (cycles)
A write on a shared line (The shared lines on other caches are invalidated)	3
A cache miss (A missed cache line is supplied by an another cache)	7
A cache miss (A missed cache line is supplied by the main memory)	22

maintained across processors through a variety of snooping and invalidation techniques. The simulated environment for this machine is described as follows.

The simulation environment consists of a functional simulator that executes parallel programs, and an architectural simulator that models the shared memory multiprocessors. An efficient program-driven simulator, MINT (Mips INTerpreter)[15] is used as a functional simulator. The MINT supplies a memory-reference generator and simulation libraries. The memory reference generator executes a program on several processors and sends an event to the architectural simulator whenever the program encounters specific operations like memory read, write or synchronization. The simulation libraries support the execution of parallel programs using a shared memory multiprocessors system and provide a set of primitives to control events received from the memory reference generator. We construct an architectural simulator based on the multiprocessors with a shared bus. Each processor is assumed to be a RISC processor with the same cache size and each instruction is executed in a single cycle except memory reference.

We assume that cache structure is 2-way set associative and that cache size is 128 Kbytes with a cache line size of 16 bytes. The simulated cache coherency protocol is the Illinois protocol.[1] On current microprocessors, the main memory access-time is about 80 ns, the clock rate is 250 Mhz (e.g., MIPS R10000, UltraSparc-II) and the system bus width is 128 bits. Table I shows timing values used in the cache coherency protocol based on these parameters including 1 address cycle and 1 bus operation cycle.

2.3. Performances

In this subsection, the performances of our benchmark parallel programs are measured across a range of grain sizes on the given simulation environment and the causes of performance variations are then

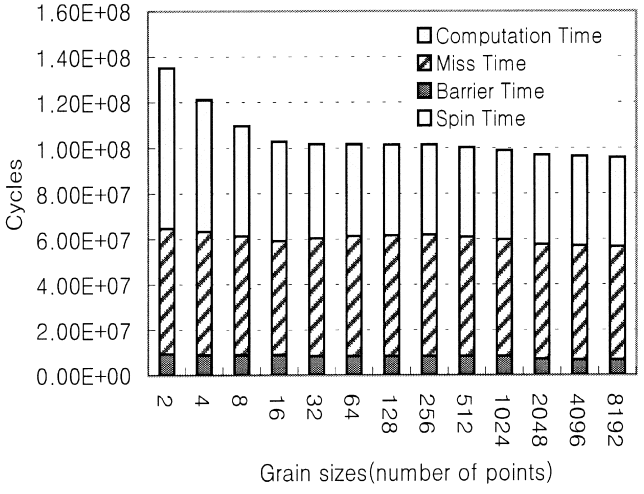

Fig. 5. FFT: 8 processors, 65536 points.

analyzed. The breakdown items in the processor execution time are composed of the spin-time, barrier-time, miss-time, and computation-time. The spin-time is the busy-waiting time due to spin-lock operations. The barrier-time is the time spent waiting at the barriers. The miss-time is the time spent waiting for data to be fetched into the cache. The computation-time is the time spent doing useful work.

• **FFT Behavior.** Figure 5 plots the breakdown of the processor execution times obtained across a range of grain sizes. This figure shows that the miss-time occupies the primary portion of the execution time. The best performance is observed with the coarsest grain size, 8192 points, since this grain size results in less miss-time and less barrier time than other grain sizes. In particular, when fine grain sizes are used, the overhead to handle fine grains incurs higher computation times.

According to our measurements, the barrier waiting times are affected by the differences of cache misses in processors, even if the same amount of work is assigned to each processor. The barrier times under fine grain sizes are higher than those at coarse grain sizes, since using the fine grain sizes raises the possibility of cache conflict misses due to the address interference between the grains allocated to the same processor. Processors that incur few cache misses reach barriers earlier than other processors, and thus they result in the increase in barrier waiting times.

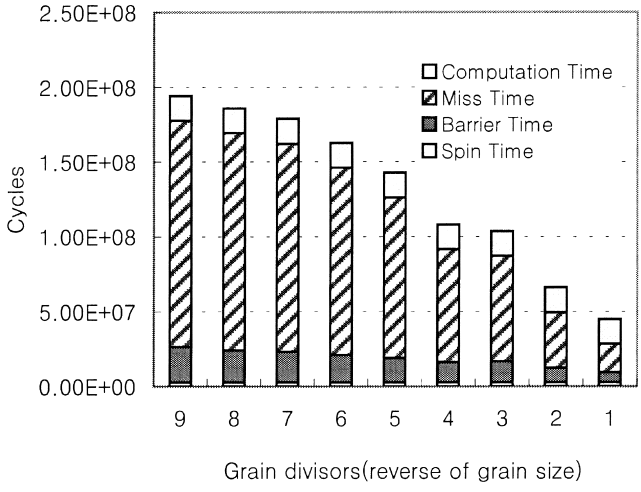

Fig. 6. LU: 8 processors, a matrix of 256×256.

• **LU Behavior.** Figure 6 shows the breakdown of the processor execution times when the LU program is run across a range of grain sizes. As described in the Subsection 2.1, the larger grain divisors mean the finer grain sizes. The best performance is from the coarsest grain size, i.e., grains divisor 1. The miss-time is given much weight in the processor execution time. The uniformity in the spin waiting time of Fig. 6 comes from the portion of the sequential component to execute the pivot rows.

The barrier waiting time of the LU program depends on not only the difference of cache misses before reaching the barrier but also the intrinsic load imbalance between processors due to gradually decreasing workload. In particular, Fig. 6 shows the higher barrier waiting time in fine grain sizes. The reason is that when using fine grain sizes, all processors are not provided with the exactly same number of grains. Even though the grain divisors mitigate this phenomenon, this imbalance is unavoidable due to the characteristic of the static scheduling policy designating the grains to each processor at the compile-time. If some processors are assigned more number of grains as compared with other processors, they arrive late at barriers. Another reason is that since fine grain sizes allocate many small grains to each processor, the address interference between the grains running on the same processor results in the difference cache misses in each processor. For these reasons, the fine grain sizes result in higher barrier waiting times.

2.4. Summary for Experimental Results

From the above simulation results, we observed that varying the barrier waiting times were highly affected by the difference of the cache misses incurred by each processor, even if nearly the same number of grains was allocated to each processor. Another hidden reason is the lack of functionality of the traditional barrier scheme. In this barrier scheme, early arriving processors cannot discover how many elements are remained uncomputed on the unarriving processors. If this information is available, for some parallel programs like our benchmark programs, the processors reaching early at a barrier can continue to execute the next iteration with the partially completed grains involved in unarriving processors.

Based on this observation, we suggest a two-phase barrier in the next Section. Though some processors may fall behind in execution, the two-phase barrier may help reduce the possibility of blocking of early arriving processors at barriers.

3. TWO-PHASE BARRIER

3.1. Basic Configuration

In the previous Section, even if approximately equal amount of work was scheduled on each processor between successive barrier synchronizations, all processors did not arrive at a barrier at the same time. The processors that are stalled waiting for other processors to reach the barrier are essentially idling and cannot do any useful work. However, when parallel programs preserve the weak data dependence between barrier synchronizations, the blind waiting at a barrier can be avoided if early arriving processors can find the information about the grains executed partially by unarrived processors. The arriving processors can continue to execute the next parallel iteration within the limited range without violating the data dependency. However, since the traditional barrier mechanism does not support this functionality to reduce the idling time of processors at barriers, we devise the two-phased barrier to provide the forward progress to early arriving processors at barriers.

The two-phase barrier is composed of two stage's individual operations rather than a specific point at which the processors must synchronize. The first stage is a *checkpoint-phase* and the second stage is a *decision-point phase*. Figure 7 illustrates the location of the traditional centralized barrier and the two-phase barrier, in the situation where a parallel program uses N processors and executes N parallel iterations, and all processors should be synchronized at the end of each iteration. As shown in this figure, the

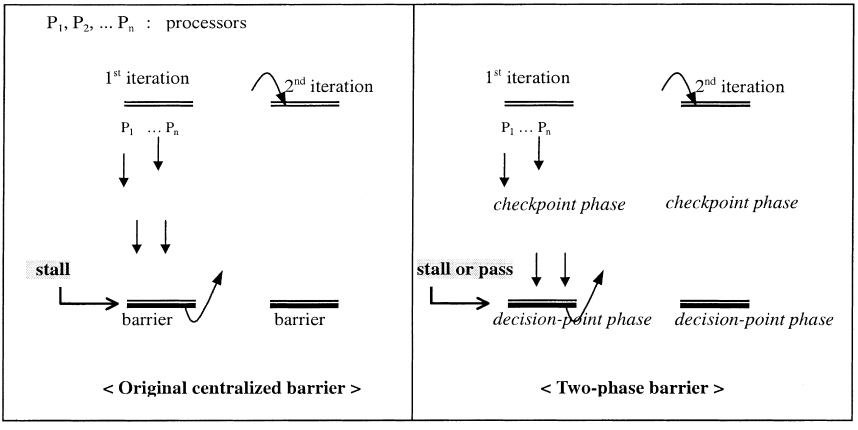

Fig. 7. The location of barriers in parallel programs.

```
checkpoint( semaphore, N)
{
  Lock(A);      /* spin lock   */
  If(checkpoint_counter = = N)
    If(decisionpoint_counter < N)
      waiting_in_checkpoint ++;
      Unlock(A);  /* spin unlock */
      P(sem_check); /* semaphore P operation */
      Lock(A);
    }                                  wake up !
  } else {                                    wake up !
    checkpoint_counter ++;
    if(checkpoint_counter = = N) /* last processor ? */
      if(waiting_processor_in_decision){
        waiting_processor_in_dicision - -;
        V(sem_decision); /* semaphore V operation */
      }
    Unlock(A);     /* spin unlock */
    goto L1000;
  }
  if(waiting_in_checkpoint) {
    waiting_in_checkpoint - -;
    checkpoint_counter ++;
    Unlock(A);       /* spin unlock   */
    V(sem_check);   /* semaphore V operation */
  }
L1000:
}
```

< checkpoint phase >

```
decision-point(semaphore, N)
{
  Lock(A);          /* spin lock */
  if(checkpoint_counter < N) {
    waiting_in_decisionpoint ++ ;
    Unlock(A);        /* spin unlock */
    P(sem_decision); /* semaphore P operation */
    Lock(A);
  } else {
    if(decisionpoint_counter == N – 1) { /* last processor ? */
      decisionpoint_counter
            = checkpoint_counter = 0;
      if(waiting_in_checkpoint) { /* waiting processors ? */
        waiting_in_checkpoint - -;
        V(sem_check); /* semaphore V operation */
      }
      Unlock(A);       /* spin unlock */
      goto L2000;
    }
  }
  if(waiting_in_decisionpoint) {
    waiting_in_decisionpoint - -;
    if(decisionpoint_counter == N – 1) { /* last processor ? */
        decisionpoint_counter = checkpoint_counter = 0;
    else
        decisionpoint_counter ++;
    Unlock(A);          /* spin unlock */
    V(sem_decision); /* semaphore V operation */
  } else {
    decisionpoint_counter ++;
    Unlock(A);
  }
L2000:
}
```

< decision-point phase >

Fig. 8. Internal structure of two-phase barrier.

ocation of the decision-point phase in the two phase barrier mechanisms is equal to that of the traditional barrier primitive, while the checkpoint phase is inserted between decision-point phases and its accurate location should be determined by programmers, based on the data dependency between barrier synchronizations.

Figure 8 shows the pseudo codes for the internal structure of two-phased barrier. Each phase exploits the spin lock primitive and two semaphore primitives to handle the global data and processors' waiting queues. And also, two global counter variables are employed in two stages. One is called as a *checkpoint counter* and the other one is called as a *decision-point counter*. Arriving at the checkpoint-phase, each processor marks its arrival by incrementing a checkpoint counter. Upon reaching at the decision-point phase, a processor increments a decision-point counter and also determines whether to stall or not, based on the value of the check-point counter. If the value of the checkpoint counter is equal to the number of processors participated in parallel processing, the processor can execute the next codes over the decision-point phase. On the other hand, the value of the decision-point counter is employed to check the stalling condition at the next checkpoint phase. For example, when a processor arrives at the next checkpoint phase over the decision-point phase, if the value of decision-point counter is less than the number of processors, this processor should be stalled at this checkpoint phase.

Figure 9 shows the flows of the synchronization and stalling events. Processors deviate from their stalling state when the last processor reaches either at the preceding decision-point phase or at the check-point phase.

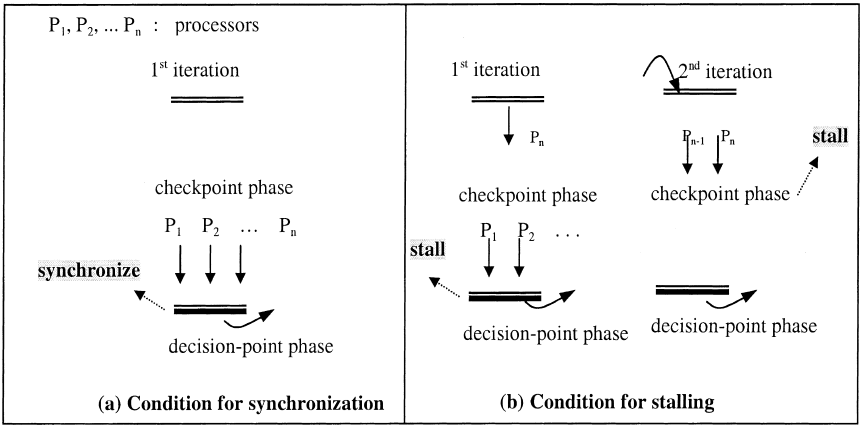

Fig. 9. Synchronization and stall of processors in two-phase barrier mechanisms.

From this functionality, the semantics of the two-phased barrier's mechanism is described in below.

• **Condition for Synchronization.** Processors are synchronized at a decision-point phase if and only if all of the processors have passed their preceding checkpoint phase.

• **Condition for Stalling.** Processors stall at a decision-point phase if and only if they have not all passed the preceding checkpoint phase. Processors also stall at a checkpoint phase if and only if they have not all reached at their preceding decision-point phase.

3.2. Examples using Two-Phase Barrier

The two-phase barrier can provide the possibility of non-blocking at a synchronization operation instead of the blind-waiting of the traditional barrier schemes. We apply the two-phase barrier to the FFT and LU programs introduced in Section 2. These programs appear to be the weak data dependency between barrier synchronizations, since the last elements of arrays are not immediately used after the synchronizations. Thus, the two-phase barrier can be applied easily.

```
1:  Procedure FFT(R, S, n)  {
2:      r = log n;
3:      m_fork( P) ;      /* set multiple processes */

4:      id = processor's label;
5:      chunk_size = grain_size × N; /* N is the number of processors used */
6:      for( m = 0; m < r –1 ; m++) { /* outer loop   */
7:          for( i = 0 ; i < n – 1 ; i++) {
8:              S[i] = R[i] ; /* address exchanging for interchanging their roles */
9:          }
10:         i = id × grain_size;
11:         for( ; i < n – 1 ; i +=chunk_size) {
12:             for( ; grain_size; grain_size --) {
13:                 if(half of each grain is executed)
14:                     checkpoint(semaphore, N);   /* checkpoint phase */
15:             /* Let(b0b1 ..... br-1) be the binary representation of   i */
16:                     j = (b0 . . . bm-1 0bm+1...br-1 );
17:                     k = (b0 . . . bm-1 1bm+1...br-1);
18:                     R[i] = S[j] + S[k] × ω (b0 . . . bm-1 0bm+1...br-1) ;
19:             }
20:         }
21:         decision-point(semaphore, N);   /* decision point phase */
22:     }
23: }
```

Fig. 10. Fast fourier transform using the two phase-barrier.

The Fig. 10 shows the case when the two-phase barrier is applied to the FFT program. From the data access pattern shown in Fig. 2, we find that if all processors execute the half of their grains, early arriving processors at the decision-point can continue to execute the next iteration without waiting to synchronize at this phase. From this observation, we insert the checkpoint phase at line 14, while the decision-point phase is at line 21. As shown in Fig. 1, the line 21 is in the same location as where the traditional centralized barrier was located.

Figure 11 shows the case when the two-phase barrier is applied to the LU program. As shown in Fig. 4, the first row among the remaining rows underneath the current pivot row is the pivot row for the next iteration. Thus, the checkpoint phase may be placed at the point where the first row within in the grain is executed. If all processors execute the first row, early arriving processors at a decision-point phase can continue to the next iteration. From this observation, we insert the checkpoint phase at line 17, while the decision-point phase is at line 21 as before.

```
1:  Procedure LU(A, n)   {
2:      m_fork( P) ;      /* set multiple processes */

3:      id = processor's label;
4:      for( k = 0; k < n –1 ; k++) { /* outer loop   */
5:          for( j = k+1; j < n – 1 ; j++) {
6:              lock(lock_var);   /* spin lock */
7:              A[k, j]   = A[k, j] / A[k, k];     /* compute a pivot row */
8:              unlock(lock_var);      /* spin unlock */
9:          }
10:         /* N is the number of processors used */
11:         grain_size = remaining rows / (N × grain_divisor );
12:         chunk_size = grain_size × N ;
13:         i = (k + 1) + (id × grain_size);
14:         for( ; i < n – 1; i += chunk_size) {
15:             for( ; grain_size ; grain_size -- ) {
16:                 if(the first row of allocated grain is executed)
17:                     checkpoint(semaphore, N);   /* checkpoint phase */
18:                 for(j = k + 1; j < n – 1 ; j++)
19:                     A[i, j ] = A[i, j] – A[i, k] × A[k, j]; }
20:         }
21:         decision-point(semaphore, N);
22:     }
23:  }
```

Fig. 11. LU program using the two phase-barrier.

3.3. Performances under Two-Phase Barrier

To evaluate the performance of the two-phase barrier, the barrier waiting times are measured for our proposed two-phase barrier scheme as well as for the traditional centralized barrier scheme. We use the same workload as described in Section 2. For detailed experiments, parallel programs are run with eight processors across the range of grain sizes described in Section 2. Since the number of processors participated affects the barrier waiting times, they are measured varying the number of processors. In particular, in the two-phase barrier, the barrier waiting time is the sum of the waiting time at decision-point phase and the one at checkpoint phase. The proposed barrier also has overhead from using *if-statements* to find the checkpoint phase within programs. Nonetheless, since the two-phase barrier provides the possibility of non-blocking synchronization, the execution time of parallel programs is expected to be improved.

3.3.1. FFT program

Figure 12(a) shows the barrier waiting times induced by both the two-phase barrier and the exiting centralized barrier across a range of grain sizes. The results show that the two-phase barrier reduces the barrier waiting times at all grain sizes. For example, the reduction rate of the barrier waiting time is 84% at the coarsest grain size (8192 points) when compared to using the traditional centralized barrier. This reduction in the waiting time contributes about 9% improvement to the total execution time. Considering the ratio of the barrier time to the total execution time

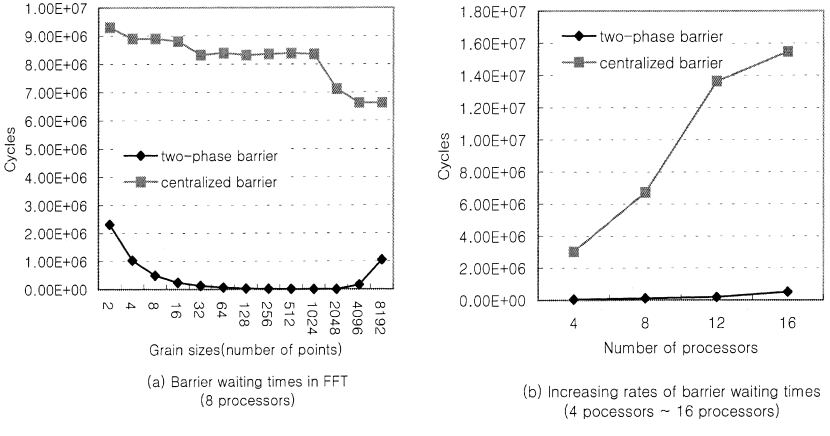

Fig. 12. Barrier waiting times in the FFT program.

shown in Fig. 5, this improvement is higher than we expected. According to our observation under using the two-phased barrier, since all processors do not proceed to the next iteration at the same time, the stall time on the shared bus for handling cache misses also deceases in the next iteration. This reduced bus waiting time also contributes to the improvement of the total execution time.

Figure 12(b) shows the barrier waiting times when the number of processors is increased. Each simulation result was measured under the coarsest grain size, since it realizes the best performance among all grain sizes as shown in Fig. 5. When the number of processors increases, the two-phase barrier enables much lower barrier waiting times than the traditional centralized barrier scheme. As an example of these measurements, when 16 processors are used, the two-phase barrier decreases the average barrier waiting time by a factor of 30.3 as compared to the traditional centralized barrier. This phenomenon results from the fact that the two-phased barrier provides non-blocking operations to early arriving processors at a decision-point phase.

3.3.2. LU program

Figure 13(a) shows the barrier waiting times of the LU program. Like for FFT, the two-phase barrier realizes lower barrier waiting times than the traditional centralized barrier. The reduction rates of barrier waiting times are from 68% at *grain divisor 1* to 37% at *grain divisor 9*, with the grain divisor 1, the reduction of the barrier waiting time improves the execution time of about 7%.

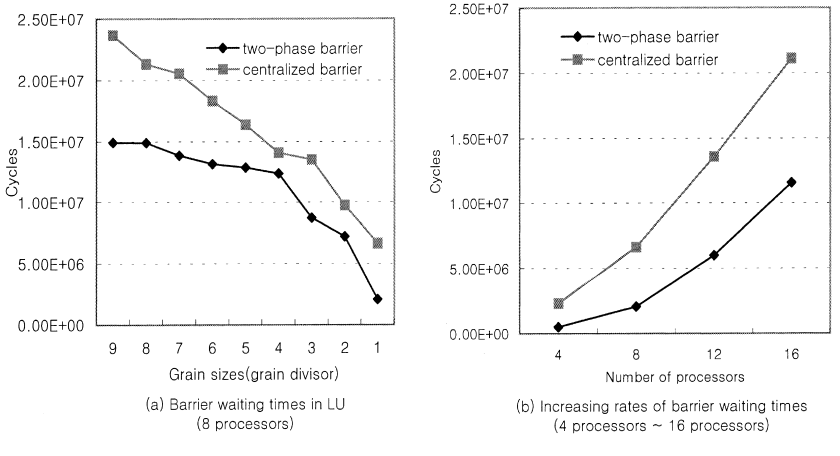

Fig. 13. Barrier waiting times in the LU program.

Figure 13(b) shows the barrier waiting times when the number of processor is increased. All results are measured under the coarsest grain size. The barrier waiting times gradually grows in both schemes as the number of processors is increased. However, the increasing rate of the two-phase barrier is less than the one of the traditional centralized barrier. With 16 processors, the two-phase barrier decreases the barrier waiting time by a factor of 1.8 as compared to the traditional centralized barrier.

4. RELATED WORKS

There are many different implementations of barriers. A representative barrier is the centralized barrier based on a global counter.[9, 10] When using the centralized barrier, each processor arriving at the barrier increments the counter and then blocks on a single, shared, completion flag. The last-arriving processor flips the flag, allowing all of the processors to then proceed. This barrier has been employed in many parallel programs because of its usable feature. But as the number of processors increases, a specific point at which all the processors must synchronize causes a high barrier waiting time. To solve this problem, several studies have shown how to increase scalability by building barriers based on a point-to-point communication among processors.[14, 12, 13] However, even with a tree-barrier,[14] the barrier waiting time increases logarithmically proportionally to the number of processors.

The topology-barrier is built based on the topology among processors.[13] This barrier needs the adjustable data structures to manage neighbor processors, and requires a large-scale modification for existing parallel programs.

Rajiv[4] suggested the fuzzy barrier to reduce the barrier waiting time. The fuzzy barrier introduced the concept of a barrier region, implying a region of instructions that can be executed by a processor while it waits synchronization. Upon reaching the first instruction in the barrier region, a processor is ready to synchronize and must synchronize before exiting the region. Processors may be executing different instructions from a specified range of instructions at the time of synchronization. This mechanism relied upon the compiler and the hardware to construct the barrier region, and focused on instruction parallelism.

In a multiprogrammed system, there were studies to explore the interaction between scheduling strategies and barrier primitives.[8, 10, 2, 6] Lim and Agarwal[8] noted that waiting times at barriers could be reduced if the program could decide whether it should wait at a barrier or be preempted based on the number of processors arriving at a barrier. However, this method needs unreasonable communication costs to track all the processors on a large-scale machine.

Markatos[10] reported that barriers using the central counter shows better performance than the tree barrier scheme, since the tree barrier had a burden to sustain the tree structures in the multiprogramming environment. They also reported that as long as processors used appropriate blocking barriers, and assuming that the time between barriers was more than several times larger than the context switching time, dedicating processors to a program was an effective scheduling policy.

Axelrod[2] suggested the combination barrier that exploits the blocking scheme among the processes on a given processor and that uses the busy waiting scheme among processors. However, this barrier had a problem that led to a uniform policy for all processes, either they all spin-wait or they all lock-wait.

Kontothanassis[6] suggested the scheduler information barrier that built on the combination barrier by using scheduler information to make an optimal spin versus block decision within processors, and then to adapt to changes at run time. A blocking synchronization was used until the number of processes reaching a barrier exceeded the number of processors in the system, whereas a busy-wait synchronization was used when the number of processes waiting at a barrier exceeded the number of processors.

In the previous works, several studies reported that the barrier waiting times were affected by scheduling policies and basic barrier structures. However, even if the same number of grains is allocated to each processor, the barrier waiting time can be affected by changes of the control paths and the cache misses. In this paper, we used detailed simulation studies to explore the interaction between the grain size and barrier waiting time in data parallel programs. On the basis of these simulation results, we suggest a new barrier primitive to enhance processor utilization by adding the new functionality to the traditional centralized barrier.

5. CONCLUSION

In this paper, to find the sources of the barrier waiting time for parallel processing, data parallel programs were executed on the various grain sizes, since the grain size affected the cache misses and the control paths, and they were assumed to be the primary sources of the barrier waiting time. The simulation results showed that even if the same number of grains was allocated to each processor in our benchmark programs, the different cache misses per processor affected the barrier waiting time more than the variation in the control paths within in the grains.

Another important issue for barrier waiting time was from the lack of functionality in the traditional barrier scheme. The main missing functionality is that processors reaching barriers can know the status of other

processors, especially when they can continue to execute the next iteration with such knowledge. To solve this problem, we suggested the two-phase barrier that comprises two stages of synchronization. This proposed scheme reduces the possibility of waiting at a barrier, since early arriving processors at a stage can continue their execution if the unarrived other processors pass through the previous stage. As a result, the two-phased barrier increases the chance of non-blocking to the processors with different arriving times at a barrier. Even if the two-phased barrier had the overhead to find the checkpoint phase in parallel programs, simulation results showed that the two-phase barrier significantly reduced the barrier waiting times and that these reduced times improved the execution times of tested programs.

In the future work, we plan to evaluate the effectiveness of using the two-phase barrier in more complex parallel programs. We will also investigate the method to detect the checkpoint phase in parallel programs by incorporating both the compiler and hardware supports.

ACKNOWLEDGMENT

This work was supported by Brain 21 Project Program by Ministry of Education & Human Resources Development and National Research Laboratory Program by Ministry of Information and Communication, Republic of Korea.

REFERENCES

1. J. Archibald and J.-L. Baer, Cache Coherence Protocols: Evaluation Using a Multiprocessors Simulation Model, *ACM Transaction on Computer Systems*, **4**(4):273–298 (November 1986).
2. T. S. Axelrod, Effects of Synchronization Barriers on Multiprocessor Performance, *Parallel Comput.*, **4**(3):129–140 (1986).
3. H. El-Rewini and T. G. Lewis, Scheduling Parallel Program Tasks onto Arbitrary Target Machines, *Journal of Parallel and Distributed Computing* (June 1990).
4. R. Gupta, The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. 3rd Int'l Conf. Architectural Support Progr. Lang. Operat. Syst.*, pp. 54–63 (April 1989).
5. I. B. Jung and J. W. Lee, Techniques for Improving the Cache Performance in Parallel Applications. In *Eleventh LASTED Int'l Conf. Parallel and Distributed Computing and Systems (PDCS'99, MIT, Boston)*, pp. 597–602 (November 1999).
6. L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, Scheduler-Conscious Synchronization, *ACM Transactions on Computer Systems*, **15**(1):3–40 (February 1997).
7. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing (Design and Analysis of Algorithms)*, The Benjamin/Cummings Publishing Company, Inc. (1994).
8. B. G. Lim and A. Agarwal, Waiting Algorithms for Synchronization in Large-Scale Multiprocessors, *ACM Transactions on Computer Systems*, **11**(3):256–294 (August 1993).

9. B. Lubachevsky, Synchronization Barrier and Related Tools for Shared Memory Parallel Programming. In *Proc. 1989 Int'l Conf. Parallel Processing*, pp. 75–179 (August 1989).
10. E. Markatos, M. Crovella, and P. Das, The Effects of Multiprogramming on Barrier Synchronization. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pp. 662–669 (December 1991).
11. C. McCann, R. Vaswani, and J. Zahorjan, A Dynamic Processor Allocation Strategy for Multiprogrammed, Shared Memory Multiprocessors, *ACM Transaction on Computer Systems*, **11**(2):146–178 (May 1993).
12. J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared Memory Multiprocessors, *ACM Transactions on Computer Systems*, **9**(1):21–65 (February 1991).
13. M. L. Scott and M. M. Michael, *The Topology Barrier: A Synchronization Abstraction for Regularly-Structured Parallel Applications*, Technical Report Technical Report 605, Department of Computer Science, University of Rochester (1996).
14. M. L. Scott and J. M. Mellor-Crummey, Fast, Contention-Free Combining Tree Barriers, *International Journal of Parallel Programming*, **22**(4):449–481 (August 1994).
15. J. E. Veenstra and R. J. Fowler, MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. 2nd Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 201–207 (January 1994).
16. T. Yang and A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors. In *The 6th ACM Int'l Conf. Supercomputing* (July 1992).

Printed in Belgium