산업기술연구(강원대학교 산업기술연구소 논문집), 제21권 A호, 2001.
Journal of Industrial Technology, Kangwon Nat'l Univ.,Korea, No. 21 A, 2001.

데이터 병렬 프로그램에서 배리어 대기시간의 분석

Analysis of Barrier Waiting Times in Data Parallel Programs

정 인 범*
Jung, In-Bum

Abstract

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 partitioned grains result in the difference in arrival time of processors at the barrier.

키워드 : *동기화, 병렬 프로그램, 캐쉬 메모리*
Keywords : *synchronization, parallel program, cache miss*

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 execute 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

* 강원대학교 전기전자정보통신공학부 교수, 공학박사

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 find the execution state of the grains running on the other processors. In our experiment, we also study the impact of the functionality in barriers.

The paper is structured as follows. In Section 2 related work is presented. Section 3 describes data parallel programs and their grain size and scheduling policies of grains used in this study. Section 4 shows the simulation environment used in this paper. In Section 5, 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. Finally, the conclusion is presented in Section 6.

2. Related Works

There are many different implementations of barriers. A representative barrier is the centralized barrier based on a global counter[8,9]. 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 [10,11]. However, even with a tree-barrier[10], the barrier waiting time increases logarithmically proportionally to the number of processors. The topology-barrier is built based on the topology among processors[12]. This barrier needs the adjustable data structures to manage neighbor processors, and requires a large scale modification for existing parallel programs.

Markatos[8] also 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.

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.

3. 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[1,2,3]. 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[3, 4].

The two data parallel programs for this study are FFT(Fast Fourier Transform) and LU(LU Decomposition)[5]. 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.

3.1 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 Example 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 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.

```
1:  Procedure FFT(R, S, n)  {
2:    r = log n;
3:    m_fork( P) ; /* set multiple processes */
4:    id = processors label;
5:    chunk_size = grain_size×N;
6:    for(m=0; m < r-1; m++) {
7:      for( i = 0 ; i < n  1 ; i++) {
8:        S[i] = R[i]; /* address exchanging */
9:      }
10:     i = id  grain_size;
11:     for( ; i < n-1 ; i +=chunk_size) {
12:       for( ; grain_size; grain_size --) {
13:         /* b0b1 ·br-1 binary rep. of i */
14:         j = (b0 . . . bm-1 0bm+1·br-1 );
15:         k = (b0 . . . bm-1|bm+1·br-1);
16:         R[i]=S[j]+S[k]×w(b0bm-1 br-1) ;
17:       }
18:     }
19:     barrier(semaphore, N);
20:   }
21: }
```

Example 1: FFT program

Figure 1 shows an example 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. 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.

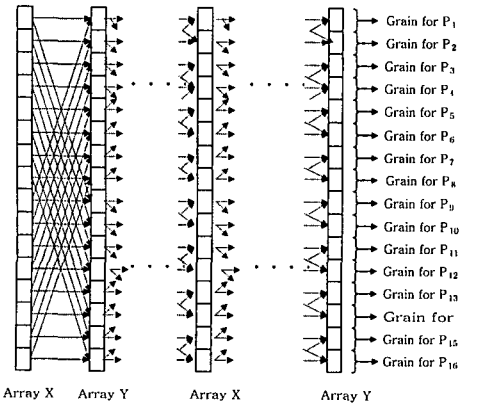


Figure 1 : Memory access patterns FFT Program program

3.2 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 Example 2 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.

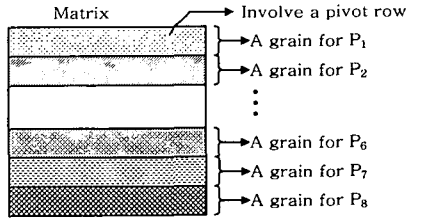
```
1: Procedure LU(A, n)  {
2:   m_fork( P) ;
3:   id = processors label;
4:   for( k = 0; k < n-1 ; k++) {
5:     for( j = k+1; j < n-1 ; j++) {
6:       lock(lock_var);
7:         A[k,j]=A[k, j] / A[k, k];
8:       unlock(lock_var);
9:     }
10:    /* N : number of processors used */
11:    grain_size=remain_rows/(N×grain_divisor);
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: }
```

Example 2: LU program

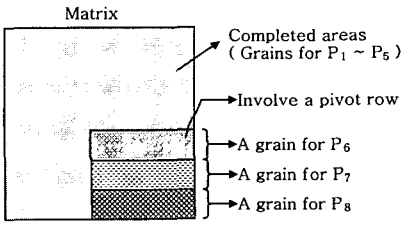
As a result, the amount of data accessed and work done per the pivot row decreases. Under the pure static grain scheduling policy, the gradual decrease in workloads appears to poor processor utilization, since the processors participating in parallel computing are not used as the computation proceeds. 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.

Under our modified static grain scheduling policy, Figure 2 shows the snapshot of the LU

program during the parallel processing. Figure 2(a) shows the initial partition of a matrix A allocated to eight processors before starting the computation. Figure 2(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 × 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.



(a) The initial partition of matrix
(P_i denotes the processor labeled i .)



(b) The partition of matrix after more than half of a matrix is computed

Firure 2: Memory access patterns in LU Program
program

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).

4. Simulation Environment

We assume the shared memory multi processors 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[6]

Cache coherency is 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)[7] 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.

Table 1: Cache timing parameters

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[8]. 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 1 shows timing values used in the cache coherency protocol based on these parameters including 1 address cycle and 1 bus operation cycle.

5. Performance Evaluation

In this Section, 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 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.

5.1 FFT program

Figure 3 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.

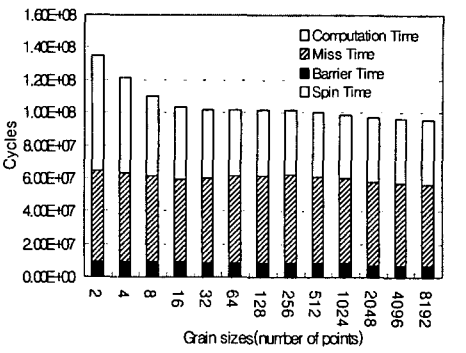


Figure 3: FFT Performance

5.2 LU program

F **Figure 3: FFT Performance** of the processor execution times when the LU program is run across a range of grain sizes. As described in the above section, the larger grain divisors mean the finer grain sizes. The best performance is from the coarsest grain size, i.e, grain divisor 1. The miss-time is given much weight in the processor execution time. The uniformity in the spin waiting time of Figure 4 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, Figure 4 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

Events (Operations)	Penalties (Cycles)
A write on a shared line	3
A cache miss	22

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.

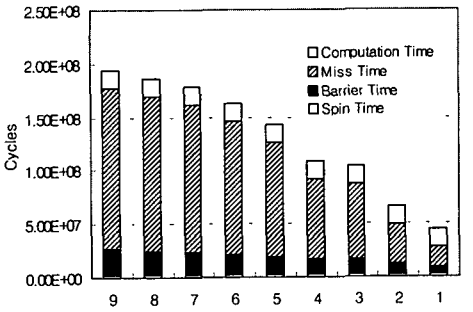


Figure 4: LU Performance

Figure 4: LU Performance

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 the barrier scheme we used, 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.

6. 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, the effective synchronization primitive will be devised in future work.

References

[1] C.McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Strategy for Multiprogrammed in Shared Memory Multiprocessor. ACM Transaction on Computer Systems, 11(2):146-178, May 1993.

[2] H. El-Rewini and T.G. Lewis. Scheduling Parallel Program Tasks onto Arbitary Target Machines. Journal of Parallel and Distributed Computing, June 1990.

[3] T. Yang and A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors, In The 6th ACM International Conference on Supercomputing, July 1992

[4] I.B. Jung and J.W.Lee, Techniques for Improving the Cache Performance in Parallel Application, In Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems, pp 597-602, November 1999.

[5] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis, Introduction to Parallel Computing(Design and Analysis of Algorithms). The Benjamin/Cummings Publishing Company, Inc., pp.179,pp.380, 1994.

[6] J.Archibald and J-L. Baer, Cache coherence protocols : Evaluation Using a Multiprocessors Simulation Model, *ACM Transactions on Computer Systems*, Vol.4, No.4, pp. 278-298, 1986.

[7] J.E. Veenstra and R.J. Fowler, MINT: A Front End for Efficient Simulation of

Shared-Memory Multiprocessors. In Proceeding of 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp 201-207, Jan 1994.

[8] Evangelos Markatos, Mark Crovella and Prakash Das, The Effects of Multiprogramming on Barrier Synchronization, In *Proceeding of Third IEEE Symposium on Parallel and Distributed Processing*, pp. 662-669, Dec 1991.

[9] B. Lubachevsky, Synchronization Barrier and Related Tools for Shared Memory Parallel Programming, In *Proceedings of the 1989 International Conference on Parallel Processing*, pp 75-179, Aug 1989.

[10] M.L. Scott and J.M. Mellor-Crummey, Fast, Contention-Free Combining Tree Barriers, *International Journal of Parallel Programming*, Vol.22,No.4, pp. 449-481, Aug 1994.

[11] J.M. Mellor-Crummey and M.L. Scott, Algorithms for scalable synchronization on shared memory multiprocessors, *ACM Transactions on Computer Systems*, Vol.9, No.1, pp. 21-65, Feb 1991.

[12] Michael L. Scott and Maged M. Michael, The Topology Barrier: A Synchronization Abstraction for Regularly-Structured Parallel Applications, Technical Report 605, Computer Science Department, University of Rochester, Jan 1996.