Problems in Characterizing Barrier Performance

by

Harry F. Jordan

Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

CSDG 88-3
October 1988

Problems in Characterizing

Barrier Performance†

Harry F. Jordan

Introduction

The barrier is a synchronization among all executing processes, all of which encounter a barrier construct at some point in their execution. The synchronization requires that all processes execute the barrier construct before any process can proceed past it to the next executable statement. It was introduced in connection with hardware support for global synchronization in the Finite Element Machine [1] and has since been used in various parallel languages [2], [3], [4] and incorporated in parallel language standards proposals [5], [6].

The barrier is usually used to satisfy a number of data dependences simultaneously by imposing sequentiality on the production and use of data items. A common usage, as suggested in the definition of a barrier appearing in Fig. 1, is to synchronize the production and use of many parts of a complex data structure simultaneously, without dealing with data items individually. The barrier is one of a class of synchronizations which can be called "generic," to indicate that processes are not identified by name. The synchronization condition is specified by the quantifier "all". An example of a synchronization which is not generic is the rendezvous, in which two specifically named processes wait

Barrier Synchronization

All processes must enter a Barrier statement before any process can complete it.

Process A	Process B	...	Process P
.	.		.
.	.		.
.	.		.
Compute part of Q	Compute part of Q	...	Compute part of Q
Barrier	Barrier	...	Barrier
Use Q	Use Q	...	Use Q
.	.		.
.	.		.
.	.		.

Figure 1: Use of the Barrier Synchronization

† This work was supported in part by the Office of Naval Research under Grant No. N00014-86-K-0204 and in part by NASA Langley Research Center under Contract No. NAS1-17070.

for each other.

There are several variations on the semantics of the barrier. Perhaps the most important is the way in which the set of processes quantified by "all" is defined. In some systems, implicit knowledge of a parallel execution environment defines the set, while in other systems a simple count of the number of processes to arrive before the barrier is satisfied is used. In the Force language [2], the semantics is modified by including a section of code between the beginning and end of a barrier construct. This code is executed sequentially by one processor after all processors have arrived at the barrier and before any process leaves it.

Implementation of the Barrier

There are numerous ways of implementing barriers on existing multiprocessors, and the alternatives have performance implications for specific architectures. A discussion of the linear versus the logarithmic organization of barrier implementations was given by Axelrod [7] while a broader set of barrier implementations was studied and measured in [8]. A number of the choices made in implementing barriers are summarized in Fig. 2. For simplicity, they are given as simple dichotomies which can be combined in various ways to yield a number of very different implementations with potentially different performance, depending on the underlying multiprocessor architecture.

The discussion will be focused primarily on shared memory multiprocessors, since the performance measurements to be studied were selected from shared memory machine examples. Most of the interprocess communication patterns for barriers can appear in either shared memory or message passing systems. The only exception is the self-scheduled updating of a shared arrival count, which is difficult in message passing machines without prescheduling at least one process to maintain the count. In the prescheduled case, processes report arrival at the barrier in some predetermined order, while a self-scheduled barrier allows processes to execute their arrival reporting code in any order. Since the barrier's purpose is to eliminate time skew between arriving processes, the situation illustrated in Fig. 3 is the normal case for a self-scheduled barrier using a critical section to update shared arrival variables. If the arrival were prescheduled so that processes had to execute their critical sections in a fixed order, say from left to right in the figure, then the arrival section of the barrier would take longer for most arrival orders.

Shared memory	---	Message passing
Prescheduled	---	Self-scheduled
Master-slave	---	Symmetric
Test&set	---	Read/write
Logarithmic structure	---	Linear structure
Distributed exit	---	Broadcast exit

Figure 2: Alternatives for Barrier Implementation

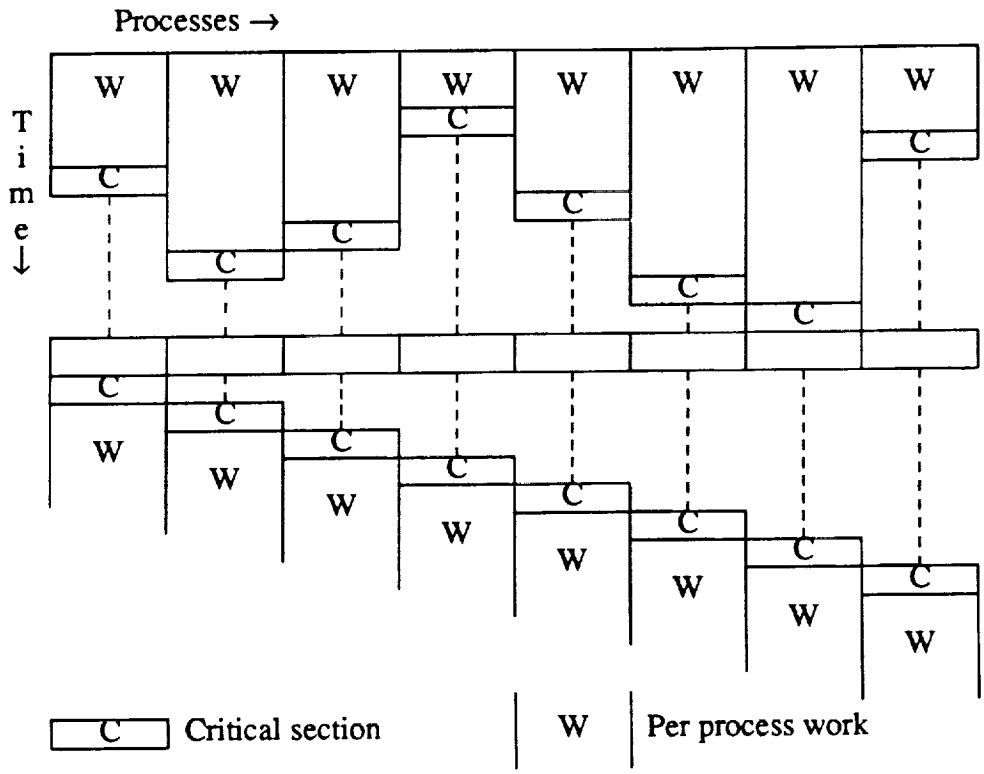

Figure 3: A Self-Scheduled Barrier with Skewed Process Arrival

The master-slave barrier structure is often associated with prescheduling, but it is perfectly possible to have the first process arriving at the barrier become the master and execute code distinct from that executed by the slaves. The characteristic of a symmetric implementation is that all processes execute the same synchronization code, which differs only in that the indices of certain synchronization variables, or the destinations of some synchronization messages may be computed from the fixed identity assigned to the process. The main advantage of the master-slave structure is that communications between the master and the slaves can be arranged so that any one synchronization variable is written by only one process. This allows the use of memory cells for which only read and write are indivisible. Symmetric implementations require something like test&set to indivisibly test and update a synchronization variable. For machines with hardware support for synchronization, the difference is small, but it can become large if all synchronizations are mediated by the operating system.

The most clear cut choice in an implementation is whether to use a linear or logarithmic pattern of communication among the processes. For systems with many physical processors, the logarithmic organization makes far better use of the system parallelism. In systems with only a few physical processors, the slightly higher computational complexity of the logarithmic structure may mean that the linear barrier can be more efficient. Depending on the nature of a machine's synchronization support, 16 processors is usually enough to make the logarithmic barrier a better choice. Figure 4 shows two

examples of logarithmic barriers. The first is the double tree barrier, which has distributed, master-slave arrival and exit phases. The second is the butterfly barrier, which is symmetric and self-scheduled, and whose arrival and exit phases are not distinct.

The choice of broadcast versus distributed exit is also dependent on the particular machine hardware. If the machine supports efficient broadcast from one source to multiple destinations, say by having one process write a variable which is read by many others, then the exit phase of a barrier may use this capability. If the barrier arrival code distinguishes a unique process, either the master or the last to arrive, this processor may broadcast a release to all other processes. For example, in the double tree barrier of Fig. 4, the inverted exit tree may be replaced by a one level broadcast from the master. Of course, if some distributed mechanism is used by the system to support the broadcast, there may be no performance gain.

Accounting for Barrier Performance

There are several different influences on time accounting in measurements of barrier performance. The major ones are:

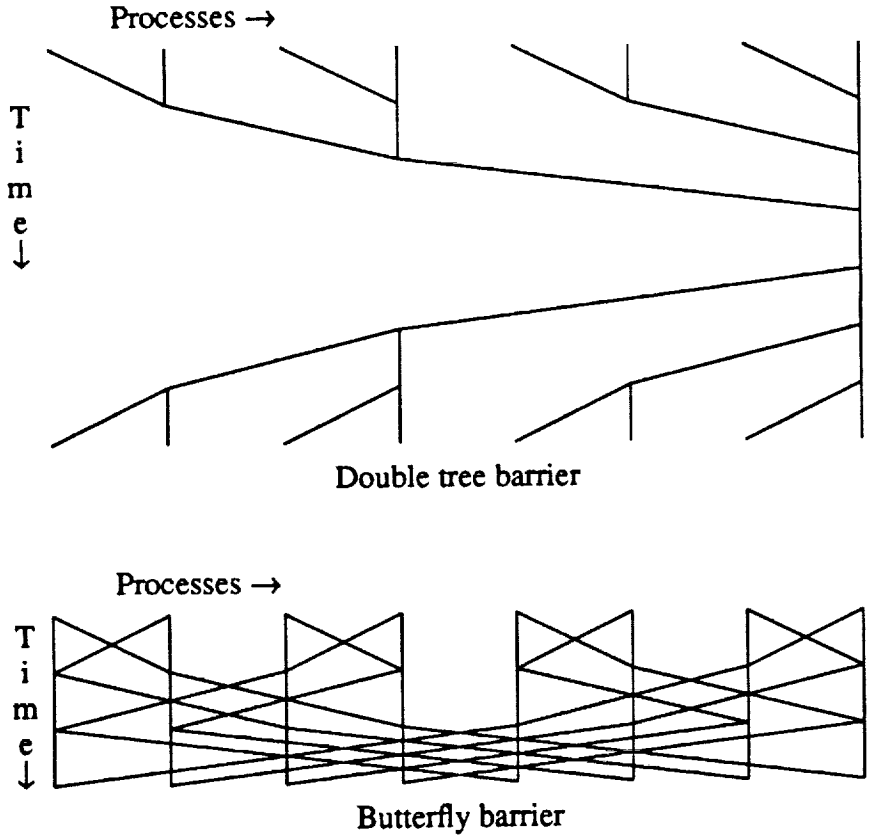

Figure 4: Two Examples of Logarithmic Barriers

delays in arrival of processes;
the waiting mechanism;
the code of the barrier implementation;
synchronization delays, i.e. critical sections;
and swapped processes, perhaps due to interrupts.

The purpose of the barrier is to synchronize processes arriving at different times, so the ideal performance is to wait for the latest arriving process. In addition to the ideal behavior, a differential delay among the arriving processes can mask some of the barrier code execution. This masking is most effective for a self-scheduled barrier which uses a critical section to update a shared count of arriving processes. If a prescheduled, master-slave barrier implementation is used, then the order of arrival of processes makes a large difference in the amount of barrier code which can be masked by the arrival time differential. Thus, in measuring the performance of such barriers, either a fixed arrival order must be specified and guaranteed, or a sufficiently large sample of random arrival orders must be taken to obtain average performance. The arrival order for self-scheduled, symmetric barriers is irrelevant, and no measurement precautions need be taken with respect to arrival order.

Once processes start arriving at the barrier, the early arrivals must be caused to wait for the later ones. The waiting mechanism may be busy waiting, virtualization of processes (swapping them out), or something intermediate between the two. Busy waiting wastes processor cycles during long delays, while virtualization is associated with an irreducible minimum overhead, which is often quite long. Intermediate positions are possible in systems which support a lightweight process model or by giving up the processor only after an initial busy waiting period, the duration of which is determined by the swapping overhead of the particular system. Our interest has been primarily in tightly coupled parallel scientific codes, so the use of process virtualization for waiting has been avoided wherever the underlying system has allowed.

The actual code executed by processes in a specific barrier implementation depends on the structure of the barrier implementation chosen, as well as on the system primitives used to implement it. The major implementation influence is the choice of linear or logarithmic barrier organization. The logarithmic organization requires somewhat more code. This can make it slightly slower than a linear barrier when used with only a few processes, but is more than compensated for by the increased parallelism possible in executing the code for even a modest number of processes. Much more important than the amount of code is the nature of the synchronization primitives on which the barrier is built. For tightly coupled parallel processing, it is important to avoid operating system overhead wherever possible. Hardware locks used to support critical sections or to implement interprocess synchronization directly are one good choice. Another is to use a master-slave implementation based on shared variables capable of atomic read and write. The time processes spend waiting for synchronization messages from other processes or for a critical section to be released is a major factor in barrier time accounting.

Instrumentation for Barrier Measurement

The most important instrument for measuring performance in a parallel system is a low overhead timer. Timers are also important in measuring sequential systems, of

course, but their effect is much easier to subtract out of a strictly serial execution history. In considering the structure of timers, there are two main aspects to take into account: the timer update mechanism and the timer sampling mechanism. The time may be updated by a mechanism which is completely transparent to the processes involved in the parallel computation. This is usually done by a hardware mechanism, but it is possible to allocate an independent process to do a transparent software update in some systems. The timer update can involve processor cycle stealing, in which case it is nearly, but not entirely transparent. It can also be done by a periodic interrupt, which usually performs other periodic operating systems functions in addition to updating a timer. Timer sampling can either be done by simply reading a shared variable, or it may require the more substantial overhead of a call to the run-time or operating system.

Another important aspect of timers for multiprocessors is whether there is a single system clock, which is accessible to all processors, or whether each processor maintains a separate hardware or software timer. Having separate timers for each processor eliminates competition when many processors try to sample the time simultaneously, but a single timer gives a more coherent measure across the processors. If it is important to keep track of the distinction between system time and user time on a per processor basis, then a timer for each processor may well be a natural choice. Systems range from having a single hardware timer which is readable by any processor as a shared memory location, as in the Encore Multimax, to having a software timer per processor, which is updated by a periodic kernel interrupt, as in the Flexible Computer Systems Flex/32.

An important parameter for a software timer is the uncertainty introduced by the time spent in the timer interrupt handler. This uncertainty can be expressed as

$$\Delta H = \frac{\text{Timer interrupt service}}{\text{Interrupt interval}}.$$

Since the timer interrupt may support periodic operating system functions of different frequencies, ΔH may vary, so that it may be appropriate to use an average value.

There are several aspects of barrier performance which may be measured. Most obvious is the effect of an ideal barrier, one with no overhead, on the behavior of a section of a parallel program. In a simple case, the effect of delaying all processes until the last one arrives can be calculated analytically, but in more complex situations, especially data dependent ones, it may be necessary to measure the effect. In addition to the ideal synchronization performed by the barrier, there are synchronization delays introduced by interprocessor synchronizing communications used to implement the barrier. Typical would be critical section delay protecting the update of a shared arrival mechanism. Finally, there are the processor cycles used to execute code associated with a particular barrier implementation. If it is assumed that barriers are the right synchronization to use for a particular parallel algorithm, then the important thing to measure is the difference between the ideal barrier behavior and that which includes the synchronization and code of the real implementation.

Given a specific parallel applications program, some simple probes of barrier behavior are possible using only an elapsed time measurement. This assumes a system dedicated to the applications program so that no substantial time is used for systems functions or time multiplexed users during the course of program execution. If the flow of program control is not altered by the violation of data dependences imposed by

barriers (only the answers are wrong), then a program can be run and timed both with and without barriers to get a measure of their total effect. Another possibility is to change the barrier implementation in a known way, say by doubling the processor cycles used in the barrier code, and to measure the effect of this change on the overall execution time. These methods are primarily useful to measure the influence of barrier synchronization on a specific parallel program to determine whether it is important to try to find a less costly synchronization method.

If measurements are made for the purpose of comparing different barrier implementations for the best performance, as was done in [8], then barriers should be measured independently of surrounding code. It is not possible to separate barriers completely from their execution environment as a result of the dependence of implementation overheads on skew and order of arrival times for different processes. A careful set of measurements on barriers with such dependence will include different types of arrival loading. A common arrival pattern occurs when two successive barriers are separated by a fixed amount of computation which is the same for each process. The order of arrival at the second barrier is then determined by the order of their release from the first one. A configuration having two barriers, separated by random, and different, amounts of work for each process, represents another useful measurement. Enough samples must be taken to average both the latest arrival time and the effects of different arrival order, if any. Another form of arrival loading which occurs commonly in practice corresponds to two barriers separated by fixed length work which contains a critical section. The time skew introduced by processes waiting to enter the critical section may influence barrier performance. For example, a self-scheduled, linear barrier does an excellent job of masking critical section skew.

The nature of the timers used to measure barrier performance is important, and in at least one case, has a drastic effect on the reliability of the measurement. Since barriers synchronize all processes, a single clock per system is most natural to their measurement. This assumes, however, that the whole system is used for the measurement. In shared or time multiplexed systems the situation is more complicated. A case presenting considerable difficulty is a dedicated system, but one which has one software timer per processor. In addition, the timer interrupts in different processors are asynchronous. The effect of the measurement uncertainty ΔH interacts with the synchronization function of the barrier in an unpleasant way. A worst case situation can occur in which all processors are interrupted while executing a barrier in a sequential order which causes the other processors to wait on the return of each interrupted processor in order to complete the barrier. Thus with P processors, the uncertainty of the measurement of barrier completion is not just ΔH but can be as large as $P \times \Delta H$. An example of this situation will be reported in the next section.

Examples

Barrier performance was one aspect of measurements performed on the Denelcor HEP shared memory multiprocessor system [9]. The system involved in the measurements consisted of four pipelined multiprocessor modules, called PEMs. Each PEM could obtain a speed of 10 MIPS (or MFLOPS) by executing 12-15 processes in parallel. The system could support up to 200 processes executing in parallel, but the pipelined

structure implied that improved performance could not result, even theoretically, for more than about 64 processes. Each PEM was equipped with a hardware performance monitor known as the System Performance Indicator, or SPI. The SPI kept track of clock cycles of 10^{-7} second along with numbers of completed instructions in several categories. The instruction categories were floating point instructions, other register to register instructions, memory reference instructions, and wave-offs (instructions which could not be issued because of synchronization).

The barrier measurements made on the HEP system were with respect to a parallel Gaussian elimination program which used barriers for its synchronization. The flow of control in Gaussian elimination is not influenced by the correctness of the floating point computations, which are the operations synchronized by the barriers. At most the time for pivoting could be influenced, but this was not the case in this program. Figure 5 shows the execution time versus number of parallel processes for the Gaussian elimination program with and without its synchronizing barriers. A summary of the results is that the synchronized version obtained a maximum speed of 7.5 MFLOPS, corresponding to 32 MIPS, while the program with barriers removed ran 9.5 MFLOPS, or 40 MIPS, which is the maximum speed for a four PEM HEP. The decrease in execution time as the number of processes increases beyond the point at which all pipelines are filled is a result of process contention for shared synchronization variables in the barriers, and of the increase in barrier complexity with number of processes.

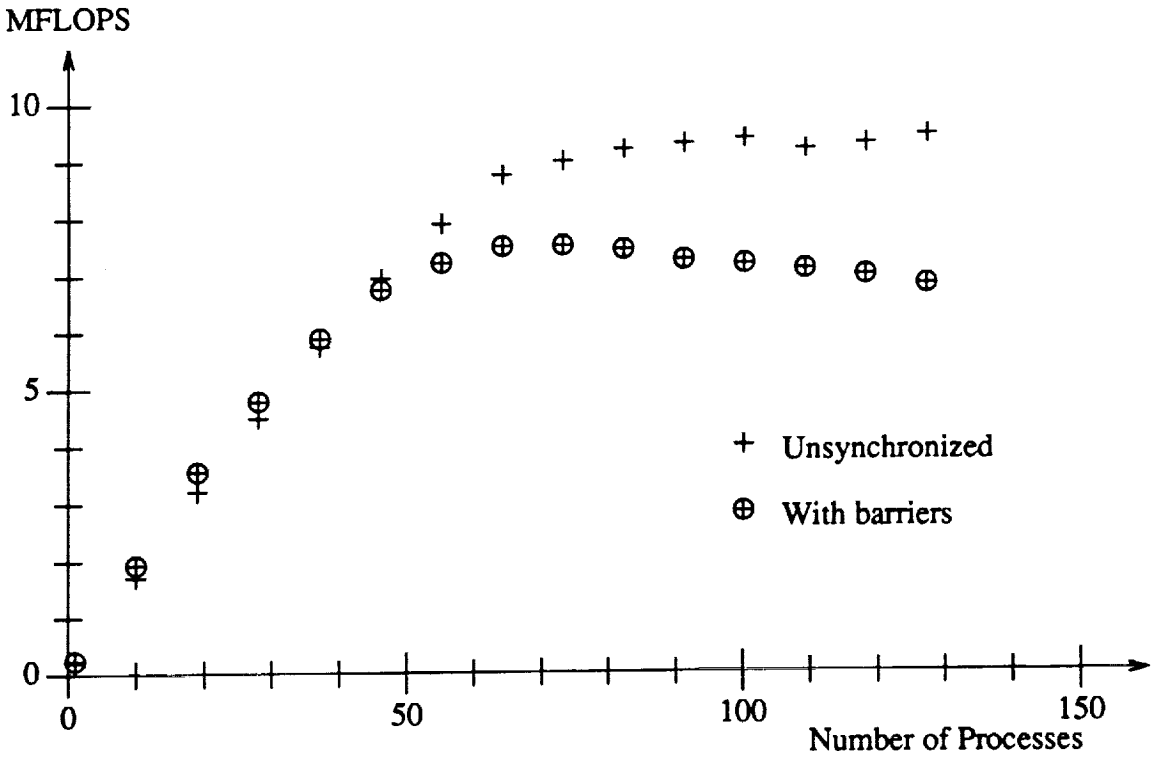

Figure 5: Barriers in Gaussian Eliminatign on a 500 by 500 Matrix

Measurements were also done by making changes in the barrier implementation. The initial implementation counted processes entering the barrier and blocked them with a single shared memory locking variable until the last process had arrived. The compiled code for this implementation amounted to about 20 instructions per process. Since four PEMs could generate memory reference retrys at four times the rate that a single memory module could handle them, it was suspected that memory access congestion made this implementation inefficient on the four PEM system. An alternate implementation of the barrier, which suspended processes as they arrived and used the last one to restart them, executed about 100 instructions per process but reduced execution time by 20%, verifying the memory contention effect. It should be noted that process suspension in this machine did not involve the operating system, but was possible with four or five user instructions. A subsequent improvement in the process suspending barrier halved the number of instructions executed per process and improved execution time by 11.8%. This allowed the determination of the fraction of execution time occupied by barrier synchronization. In the final Gaussian elimination program about 14% of the time is spent in barrier synchronization. This illustrates how program specific changes in a well understood code can be used to examine barrier performance.

An extensive set of measurements was done by Arenstorf [8] on the Flexible Computer Corporation's Flex/32 running under the MMOS [10] operating system. This system consists of 20 single board microprocessors associated with a combination of shared and private memories. The particular system used allocated two processors to running in a single processor mode, leaving 18 processors available under MMOS. The experiments were run with a fixed mapping of one process per processor with no multiprogramming. The operating system is distributed over the processors, and in particular, has a software timer per processor. The measurements compared several different implementations of the barrier for performance in different environments, but of most interest here is the effect of the software timers running asynchronously on each processor.

The Flex/32 system measurements exhibit the problem, mentioned earlier, of multiplying the measurement uncertainty ΔH due to the service time for timer interrupts by the number P of processes. The standard configuration of the MMOS operating system uses a 20 millisec. interrupt interval with a service time of 0.3 ms. The resulting $\Delta H = 1.5\%$ is acceptable for timing single stream phenomena, but using 18 processors synchronized by barriers, it presents a significant problem. In the worst case sequence of timer interrupts using all 18 processors, some processor is unavailable to satisfy the barrier synchronization 27% of the time. The resulting uncertainty in barrier measurements is not a result of inaccurately sampling the value of the time but of overheads involved in updating the time. The problem of obtaining accurate times was solved by increasing the timer interval to one second, thus reducing ΔH to 0.03% and the worst case influence on barrier measurements to 0.54%. Of course, it was necessary to make very long timing runs to reduce the effect of the one second accuracy of the time to an acceptable percentage of the measurement.

A final measurement of barrier performance is of interest in demonstrating an extreme case of the effect of all processors not being available to execute the barrier simultaneously. In this case, it is not timer interrupts which occupy the processors but multiprogramming activity. The measurements were performed by Benten [11] on the Sequent Balance 8000, a bus connected, shared memory multiprocessor. This system

had eight processors, organized two per board sharing a cache memory. All memory is shared and is connected to the bus, so that it is uniformly accessible to all processors. The version of the system to which we had access ran a Unix style operating system which was highly multiprogrammed and treated processors as a schedulable resource. We had no way of locking processes to processors other than ensuring that there was no other load on the system, and no reason in our own program to time multiplex processors.

A barrier synchronized Gaussian elimination program, similar to the one measured on the HEP, was run for varying numbers of processes. The character of the results is not surprising, but the magnitude is interesting. As shown in Fig. 6, when the program was run on an unloaded system, normal speedup results were observed until multiprogramming was forced to occur by virtue of the number of processes exceeding the number of physical processors. With nine processes time multiplexed on eight processors, the execution time is significantly longer than that for one process. This reflects the fact that all processes must be coscheduled for a barrier which is implemented independently of the operating system to perform reasonably. An alternative to the barrier must be used for synchronizing parallel programs unless the operating system's scheduler can cooperate with the barrier construct. Figure 7 shows that the presence of multiprogramming load on the system merely causes the effect to be seen for fewer processes and to increase in magnitude.

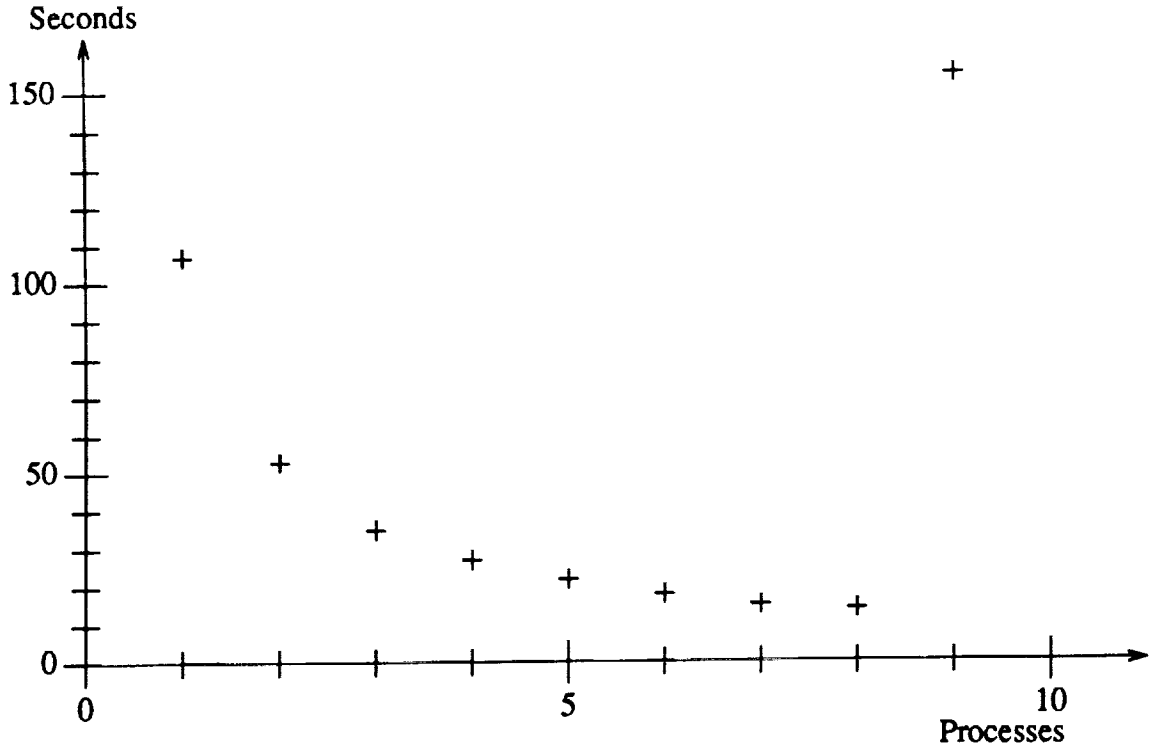

Figure 6: Effect of Lack of Coscheduling, Single Program

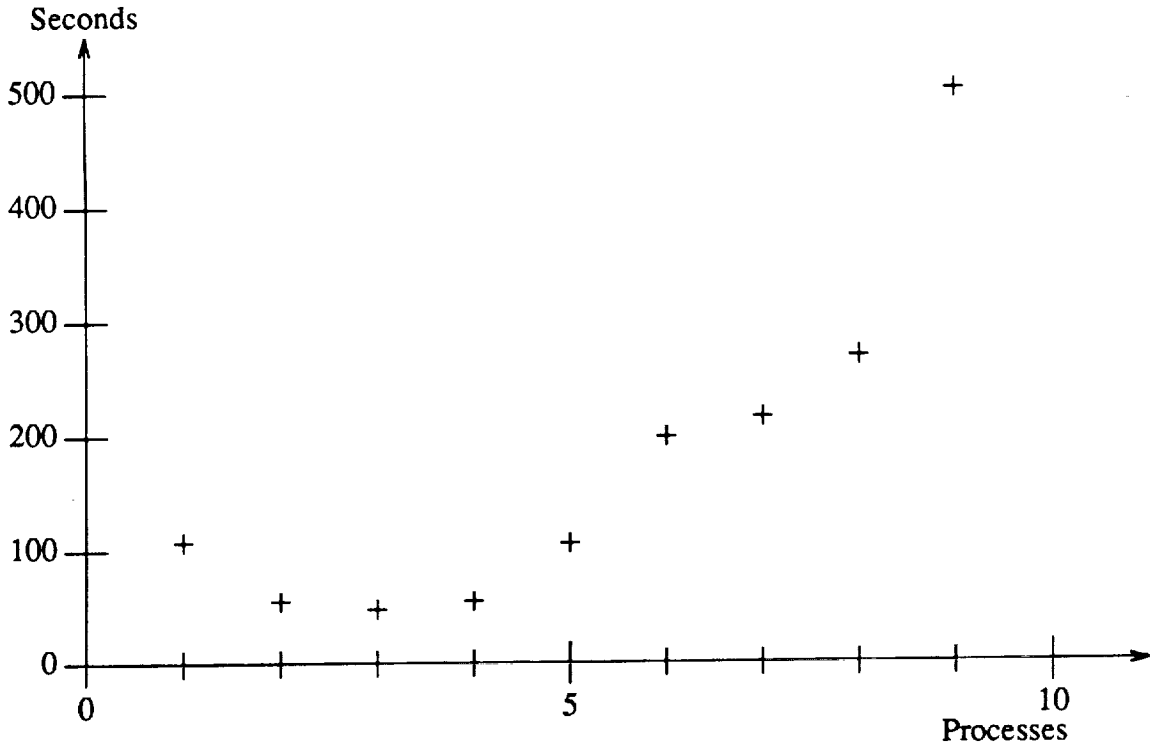

Figure 7: Effect of Lack of Coscheduling, with Multiprogramming Load

Conclusions

The barrier is a convenient synchronization mechanism in multiprocessors, especially of the shared memory type. Measuring the performance of barriers may be used to determine their effect on the performance of a parallel program, in order to optimize the program, and to choose the best method and arrangement of synchronization. Measurements can also help select between different types of barrier implementation, to suit it to the machine environment. The measurement of barrier performance offers some unique difficulties, which result from the global nature of the synchronization. All processes must be available for the barrier to complete, and any virtualization of processes will have a noticeable effect. Another problem is to separate overhead due to a specific barrier implementation from the waiting time imposed purposely on processes by the nature of the synchronization. Hardware or software support for measuring the waiting time of processes directly would significantly aid barrier measurement. In the work done to date, waiting times must be inferred from several indirect time measurements.

References

[1] H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," *Proc. 1978 Int'nl Conf. on Parallel Processing,* IEEE Computer Society Press (1978) pp. 263-266.

[2] H. Jordan, "The Force," in *The Characteristics of Parallel Algorithms,* L. Jamieson, D. Gannon and R. Douglass, Eds., Chapter 16, MIT Press, Cambridge, MA (1987).

[3] E. Lusk and R. Overbeek, "Use of Monitors in Fortran: A Tutorial on the Barrier, Self-scheduling Do Loop and Askfor Monitors," *Argonne National Laboratory Report No. ANL-84-51,* Argonne, IL (1985).

[4] A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems,* Sequent Computer Systems, Inc., Beaverton, OR (1985).

[5] P. Frederickson, R. Jones and B. Smith, "Synchronization and Control of Parallel Algorithms," *Parallel Computing,* V. 2, No. 3 (1986) pp. 265-254.

[6] M. Furtney, R. Kuhn, B. Leasure and E. Plachy, "PCF Fortran: Language Definition," Parallel Computing Forum, Kuck & Associates, 1906 Fox Drive, Champaign, IL 61820, Version 1 (Aug. 16, 1988).

[7] T. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," *Parallel Computing,* V. 3, No. 2 (1986) pp. 129-140.

[8] N. S. Arenstorf and H. F. Jordan, "Comparing Barrier Algorithms," ICASE Rept. No. 87-65, NASA Langley Res. Ctr., Hampton, VA, Sept. 1987, to appear in *Parallel Computing.*

[9] H. F. Jordan, "Performance and Program Structure in a Large Shared Memory Multiprocessor," in *New Computing Environments: Parallel, Vector and Systolic,* Arthur Wouk, Ed., pp. 201-217, SIAM, Philadelphia, PA, 1986.

[10] *Multicomputing multitasking operating system (MMOS) reference manual,* Flexible Computer Corporation, Dallas, TX (1986).

[11] M. S. Benten and H. F. Jordan, "Multiprogramming and the Performance of Parallel Programs," *Proc. 3rd SIAM Conf. on Parallel Processing for Scientific Computing,* Los Angeles, CA (Dec. 1987).

BIBLIOGRAPHIC DATA SHEET	1. Report No. ECE Tech. Rept. 88-1-4	2.	3. Recipient's Accession No.
4. Title and Subtitle Problems in Characterizing Barrier Performance			5. Report Date October 1988 6.
7. Author(s) Harry F. Jordan			8. Performing Organization Rept. No. CSDG 88-3
9. Performing Organization Name and Address Computer Systems Design Group Department of Electrical and Computer Engineering University of Colorado Boulder, CO 80309-0425			10. Project/Task/Work Unit No. 11. Contract/Grant No. ONR N00014-86-K-0204
12. Sponsoring Organization Name and Address Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000			13. Type of Report & Period Covered Interim 14.

15. Supplementary Notes Also supported in part by NASA Langley Research Center under NASA Contract Number NAS1-17070

16. Abstracts The barrier is a synchronization construct which is useful in separating a parallel program into parallel sections which are executed in sequence. The completion of a barrier requires cooperation among all executing processes. This requirement not only introduces the "wait for the slowest process" delay which is inherent in the definition of the synchronization, but also has implications for the efficient implementation and measurement of barrier performance in different systems.

Types of barrier implementation and their relationship to different multiprocessor environments are described. Then the problem of measuring the performance of barrier implementations on specific machine architectures is discussed. The fact that the barrier synchronization requires the cooperation of all processes makes the problem of performance measurement similarly global. Making non-intrusive measurements of sufficient accuracy can be tricky on systems offering only rudimentary measurement tools.

17. Key Words and Document Analysis. 17a. Descriptors

Multiprocessors
Synchronization
Performance Measurement
Barriers

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 12
	20. Security Class (This Page UNCLASSIFIED	22. Price