Generalized Dynamic Group Barrier Synchronization

S. K. S. Gupta and D.N. Jayasimha

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email: sandeep@cis.ohio-state.edu, jayasim@cis.ohio-state.edu.

Abstract

This paper presents a process based dynamic barrier scheme which can be implemented on large scale shared memory architectures. It improves upon the dynamic barrier synchronization scheme based on group locks proposed by Dimitrovsky [5]. The set of processes which meet at a barrier is determined from the semantics of the program. The main idea behind the scheme is to identify the dynamic set of processes (the *interested* processes) which need to barrier synchronize at various control points in the execution of the program. This identification, which is begun at compile time is completed at run time aided by the information generated at compile time. Compared to the other schemes presented in the literature, this scheme does not delay the execution of processes which need not synchronize at a barrier (the *uninterested* processes). Since a minimal set of processes synchronize at a barrier, the synchronization overhead, which grows superlinearly with the number of processes, is less compared to the other schemes. This scheme also exploits the use of concurrent barriers at a finer level of granularity than all other previous schemes.
Keywords: shared memory multiprocessing, process based barrier synchronization, dynamic group synchronization, parallelizing compilers, control flow analysis.

Contents

1 Introduction

A barrier is a global synchronization point in the control flow of a program. Many hardware, software, and hybrid schemes have been proposed in the literature to efficiently implement barrier synchronization. Hardware barrier schemes [2, 8, 11, 12, 16, 17] are fast and they are processor based[1]. They use special purpose hardware- for example, atomic lock memory in the Sequent Balance system [18], concurrency bus in the Alliant FX-8 [4], and combining network in the Ultracomputer [9] and the IBM RP3 [19]. Some machines are designed for the purpose of achieving fast barrier synchronization- for example, the Barrier MIMD machine [16, 17]. These hardware schemes have the disadvantage that they all support the processor based view of barrier synchronization. Most of the hardware schemes are static since the group of processors meeting at a barrier should be known at the compile time. Time-constrained barrier [8] which is a dynamic barrier scheme is also processor based. For a multiuser, multiprogrammed environment, a barrier synchronization scheme should support a process based view and should support concurrent barriers (i.e., should support two or more barriers to be simultaneously active). Beckmann and Polychronopoulos [2] have proposed a fast barrier synchronization hardware which can be adapted to multithreaded processors. But the barrier hardware puts a limitation on the number of threads per application. This limitation can be more restrictive than the restriction put by the operating system and the multiprocessor itself. The scheme also supports concurrent barriers. The processors (processes) that *need* to synchronize at a barrier B are called the *interested* processors (processes) and those that need not are called the *uninterested* processors (processes) with respect to barrier B. Most hardware schemes use masking to block out the uninterested processors which are not interested in participating in the barrier.

Software schemes [7, 10, 15], on the other hand, are process based. One important advantage of process based barrier synchronization schemes is that they function correctly even if processes migrate from one processor to another. The reasons for process migration from one processor to another may be to achieve better load balancing and to take care of processor failures.

The main drawback of software schemes is their poor performance. In general, performance of processes which do not participate in barrier synchronization should not be affected by other processes meeting at a barrier. In hybrid barrier synchronization schemes like that proposed by Birk, et al. [3], for example, the compiler introduces dummy barrier instructions in the code of the uninterested process. Such dummy instructions could result in the slowing down of the uninterested processes and lead to degradation in performance. Further, there is no reason[2] why a process should be slowed down due to barriers in which it need not participate. Placement of dummy instructions can also adversely affect the deadlock free property of the program. Hence the introduction of the dummy barrier instruction by the compiler will makes proving correctness of the program more difficult. Further, since the cost of a barrier grows superlinearly with the number of processes meeting at a barrier [1], the uninterested processes (unnecessarily) contribute to a degradation in the performance.

Most of the concurrent barrier schemes mentioned above impose an unnecessary ordering on the concurrent barriers which can be executed in parallel. The ordering may be imposed due to

[1] A barrier scheme is processor(process) based if processors(processes) are the logical entities which synchronize at a barrier. When we refer to the term "processes," we always mean processes belonging to a single program.

[2] except for simplifying the implementation

the placement of a dummy barrier instruction or the loading of the barrier mask in the hardware register. This may increase the overall waiting time for a barrier in the system and hence lengthen the total execution time.

Masking in hardware barrier schemes and the introduction of dummy instructions in the software barrier schemes are consequences of the fact that these barrier schemes require that the number of processes participating in the barrier be known statically. To overcome this limitation, Dimitrovsky [5] developed a mechanism for processes to form groups dynamically based on group locks [14], which then can meet at a barrier. Gottlieb, et al. [7] have implemented this dynamic group barrier scheme using the Fetch & Increment instruction. In this scheme, the interested processes form a group by registering to that group and then meet at a barrier. A process can leave a group whenever it wishes to do so. One limitation of the group barrier scheme is that the criterion for determining both the group membership and the size of the group is arbitrary- the group leader decides to close a group based on a timeout mechanism. This may be efficient but it is not flexible. For example, such a mechanism cannot handle situations where the group as a whole should satisfy some consistency criterion. In Section 7 we give an example of parallel garbage collection program in a thread-based programming environment where the group should be closed under pointer chasing.

This paper presents a process based dynamic barrier scheme which can be implemented on large scale shared memory architectures. It improves upon the dynamic barrier synchronization scheme based on group locks proposed by Dimitrovsky. The set of processes which meet at a barrier is determined from the semantics of the program. The main idea behind the scheme is to identify the dynamic set of processes (the *interested* processes) which need to barrier synchronize at various control points in the execution of the program. This identification, which is begun at compile time is completed at run time aided by the information generated at compile time. Compared to other schemes this scheme does not delay the execution of processes which need not synchronize at a barrier (the *uninterested* processes). Since a minimal set of processes synchronize at a barrier, the synchronization overheads, which grows superlinearly with the number of processes, are less compared to the other schemes. This scheme also exploits the use of concurrent barriers at a finer level of granularity than all other schemes.

In the next section we define the generalized dynamic group barrier (GDGB) scheme. Section 3 describes the GDGB mechanism. In Section 4 we discuss some implementation issues. In Section 5 we suggest an architecture to efficiently implement the GDGB scheme and also outline the implementation of the scheme on a BBN Butterfly like architecture. In Section 6 we compare the performance of GDGB scheme with other barrier schemes. Section 7 gives some examples where this scheme would be particularly useful and efficient. In Section 8, we summarize the main features of the scheme and suggest possible future directions this research could take.

2 Generalized Dynamic Group Barrier Synchronization

A logical (maximal) set of processes all of which or subsets of which have to barrier synchronize (possibly at different control points in the program) is associated with each barrier identifier. This logical set, which could be formed completely only at run time, is determined from the semantics of the program. A minimal set of processes which have to barrier synchronize is associated with a

```
A(j) : Barrier; /* A is barid and j is the associated barrier instance variable */
doall i = 1 to n
  j := 1; /* A barrier control point */
  Barrier(A,j);
  do k = 1 to 2;
    if (cond) then {
      if (Odd(j)) then
        j= j+1 /* A BCP */
      else j = j+2; /* A BCP */
      /* Program Fragment */
      Barrier(A,j);}
    else {
      if (Odd(j)) then
        j = j+2 /* A BCP */
      else j = j+1; /* A BCP */
      /* Program Fragment */
      Barrier(A,j);}
  enddo ;
enddoall.
```

Figure 1: An Example

particular value of the variable k. This set of processes is itself a subset of the processes associated with the barrier identifier A. Such a dynamic group of synchronizing processes is represented by a two tuple- *(A,k)* where A is the *barrier identifier* and k is an integer variable called the *barrier instance variable*.

The reasons why we represent barriers in this seemingly involved manner are threefold:

- This representation helps us in identifying the minimal set of processes (the *interested* processes) that need to barrier synchronize. Two consequences of this identification are that barrier overheads are lower in our scheme and the *uninterested* processes are not delayed because a related set of interested processes need to barrier synchronize.
- This representation helps us in building the interested process set using a combination of static and runtime analysis.
- Two or more different instances with the same barrier identifier could be active simultaneously if the set of processes involved with each instance is disjoint from the other. Consequently, this scheme exploits the use of concurrent barriers at a finer level of granularity than all the other schemes proposed in the literature. In the other schemes, using our terminology, only two or more barriers each with a different identifier can execute concurrently while in our scheme two or more instances of the *same* barrier can execute concurrently. The parallel merge sort example given in Section 7 illustrates the use of concurrent barrier instances.

The condition for completion of a GDGB synchronization is,

Definition 2.1 (Condition for synchronization at the $\mathbf{i}^{th}$ instance of barrier A) *The processes have synchronized at the* $\mathbf{i}^{th}$ *instance of barrier A iff the following conditions hold*

1. *No process henceforth reports at* **Barrier(A,k)** *with value of* $\mathbf{k} = \mathbf{i}$.
2. *All the processes which have executed* **Barrier(A,k)** *with the value of* $\mathbf{k} = \mathbf{i}$ *have reported.*

Associated with each barrier instruction *Barrier(A,k)* there is an associated assignment statement which sets the value of the barrier instance variable k to the instance number of A at which the processes wish to synchronize. We will refer to this assignment statement as a **barrier control point**(BCP). Any barrier control point associated with a barrier instruction *Barrier(A,k)* satisfies the following restrictions:

1. **Value restriction:** A BCP can only increment the value of the barrier instance variable.
2. **Placement restriction:** After the execution of a BCP, the associated barrier instruction should be guaranteed to execute.

The **value restriction** ensures that a process does not synchronize at the same instance of the barrier twice and a process synchronizes at barrier instances in a strict monotonically increasing order. As we will see latter the **placement restriction** permits an efficient implementation of our proposed scheme.

The need for a dynamic group barrier is illustrated through the program segment showed in Figure 1. This example also illustrates the use of the generalized group barrier. The program segment initially creates n processes which then meet at the first instance of barrier A. After that processes which satisfy the condition $cond$, meet at an even numbered instance of barrier A. Otherwise, processes meet at an odd instance of the barrier. Note the dynamic nature of a process set meeting at a particular instance of barrier A. It is some subset of the process set $\{P_1, P_2, \ldots, P_{n-1}, P_n\}$ depending on the value of the condition, $cond$, where each $P_i : 1 \leq i \leq n$ is a process executing the i^{th} iteration of the doall loop.

3 GDGB Mechanism

The main problem in any dynamic barrier scheme is to determine at runtime the process set of a particular barrier (i.e., the exact set of processes which need to barrier synchronize). To aid in this determination, the GDGB mechanism consists of the following three phases:

1. **Process set recognition phase:** In this phase the set of processes which synchronize at the current instance of the particular barrier is determined. The process recognition phase is completed only at run time.

2. **Barrier recognition phase:** The barrier recognition phase completes only when all the processes that have been recognized in phase 1 have reported their process completion.

3. **Wakeup/Notification phase:** The notification phase consists of notifying the waiting processes that the barrier is completed. This phase begins after all the interested processes have arrived.

The highly flexible nature of this barrier scheme demands a sophisticated parallelizing compiler (see [20] for examples of such compilers) which generates information used at runtime to predict the process set of a barrier instance as early as possible. In many situations the process set can be statically determined. But in situations like the program given in Figure 1, the exact process set cannot be computed by the compiler. The barrier mechanism described below will use the compiler generated information to compute the set of processes which are definitely going to participate and those which are likely to participate in the barrier. Since this scheme is process based, it does not suffer from any of the drawbacks of barrier based schemes which use masks in hardware or dummy instructions in software (the drawbacks of such schemes were cited in Section 1).

At compile time the set of all processes can be partitioned into the following three sets with respect to a barrier A and its i^{th} instance:

1. **Sure_Set:** SS(A,i) is the set of processes which will synchronize at the i^{th} instance of the barrier A.

2. **Probable_Set:** PS(A,i) is the set of processes which will probably synchronize at the i^{th} instance of the barrier A.

3. **Uninterested_Set:** US(A,i) is the set of processes which will not synchronize at the i^{th} instance of the barrier A.

If PS(A,i) $= \phi$, then the process set recognition phase can use the statically generated sure_set information and the GDGB synchronization is similar to the regular barrier synchronization. If PS(A,i) $\neq \phi$ and SS(A,i) $\neq \phi$, then the process set recognition and barrier recognition[3] phases can be done concurrently. All the known barrier techniques do not make a distinction between the Sure and the Probable sets. The processes belonging to the set $(PS \cup SS)$ have to participate in the barrier. In the proposed GDGB scheme, however, the PS set is partitioned at runtime into the following two sets:

- **PS_U:** Compile time unsure, runtime uninterested,
- **PS_S:** Compile time unsure, runtime interested.

Only the processes in the set $(PS\_S \cup SS)$ participate at a barrier. To determine if a process $p \in PS(A,i)$ is interested in participating at the i^{th} instance of barrier A, the compiler associates with each control point in the program a set called the future barriers set (FBS). We now present definitions needed to understand the GDGB mechanism.

[3] We can use the adaptive combining tree proposed by Gupta and Hill [10] to speedup the second phase.

Definition 3.1 (Control point) *A node in the control flow graph is called a control point if it contains at least one BCP and each BCP in the node is associated with a distinct barrier.*

Definition 3.2 (Future Barrier Set (FBS)) FBS(c) *is the set of all possible future barriers that can be executed after the control point c. Formally,* **FBS(c)** *is a subset of the cross product of the set B, the set of barrier names in the program, and N, the set of natural numbers, i.e. $FBS(c) \subseteq B \times N$, with the following restrictions,*

- *if a tuple $(A, l) \notin FBS(c)$, then the process $p, (p \in US(A, l)$, which reaches the control point c is guaranteed to not participate in the l^{th} instance of the barrier A.*
- *if a tuple $(A, l) \in FBS(c)$, then the process $p, (p \in PS(A, l))$, which reaches the control point c is likely to participate in the l^{th} instance of the barrier A.*

Definition 3.3 (Active control point,ACP(p,t)) *Given a process p and a time instant t, the active control point $ACP(p, t)$ is the last control point executed by p.*

Definition 3.4 (Active instance number, AIN(A,p,t)) *Given a process p, a time instant t and a barrier name A, the active instance number AIN(A,p,t), is the value of the instance variable associated with barrier A at the active control point ACP(p,t).*

Definition 3.5 (Future Barrier Set restricted to a barrier instance)
The set $FBS(c)/(A, i) = \{(A, b) | \; (A, b) \in FBS(c) \wedge b \geq i\}$, is called the FBS of control point c restricted to i^{th} instance of barrier A.

Observation: As can be easily verified, a process $p \in PS(A, i)$ does not participate in the i^{th} instance of the barrier A if either its active instance number corresponding to barrier A is greater than i or the tuple (A, i) is absent from the FBS set of some active control point.

Theorem 3.1 (Condition_PSU) *A process $p \in PS(A, i)$ does not participate in the i^{th} instance of the barrier A if the condition*

$$(AIN(A, p, t) > i) \vee ((AIN(A, p, t) \leq i) \wedge ((A, i) \notin FBS(ACP(p, t))/(A, AIN(A, p, t)))) \quad (1)$$

holds at some time instant t, before the i^{th} instance of the barrier A has completed.

Proof: The proof follows from the **value restriction** and the **placement restriction** on a BCP, the definition of the FBS, and the above observation. □

Theorem 3.2 (Condition_PSS) *A process $p \in PS(A, i)$ participates in the i^{th} instance of the barrier A if the following conditions hold*

1. *$AIN(A, p, t) = i$.*

```
procedure Determine_size(A, a)
    /* A is a barrier name and a is its instance number*/
    PS_U(A, a) = φ;
    PS_S(A, a) = φ;
    for each p ∈ PS(A, a) in parallel
      repeat
        if (Condition_PSU) then {PS_U(A, a) = PS_U(A, a) ∪ {p}; break;}
        if (Condition_PSS) then {PS_S(A, a) = PS_S(A, a) ∪ {p}; break;}
      until false;
    endfor
    Barrier_size(A,a) = |PS_S(A, a)| + |SS(A, a)|;
    Barrier_proceed(A,a) = true;
end .
```

Figure 2: Psuedocode for determining size of the PS_S set

2. *There is no other control point between $ACP(p, t)$ and the barrier instruction Barrier(A,k) which modifies k.*

3. *The instruction Barrier(A,k) is guaranteed to be executed.*

Proof: The proof follows obviously from the **value** and **placement** restrictions on a barrier control point.
□

For each active barrier (A, a) the runtime component of the GDGB mechanism should determine the exact size of the process set of barrier (A, a). Procedure *Determine_size* given in Figure 2 uses theorems 3.1 and 3.2 to determine sizes of the dynamic sets PS_S and PS_U The procedure *Barrier_size* is used to record the size of the barrier (A, a) and the boolean *Barrier_proceed* is used to indicate that the process set recognition phase is completed. After this the barrier mechanism can now proceed as it would have proceeded had the process set been determinable.

4 Implementation Issues

In this section we consider two important issues from the implementation point of view. The first issue is concerned with the placement of the BCPs and the second with the computation and the storage of the FBS. In doing so we show that appropriate compiler techniques exist to make an implementation of GDGB feasible. In the next section we describe one such implementation.

Placement of Barrier control points

The barrier control point for a barrier should be placed as early as possible in the program. This will overlap the process set recognition phase for a barrier with other useful computation. A BCP can be placed at any point before the actual barrier synchronization point where the placement restriction is not violated For doing this standard compiler techniques can be used. For example, a BCP B, can be placed in any dominator n_j of the node n_i which contains the barrier instruction $Barrier(A,k)$, corresponding to BCP B, provided there is no other barrier instruction involving A along any path from the dominator n_j to the node n_i in the control flow graph. All the paths from a dominator of a node, n, to the end node of the control flow graph passes through n. Hence this will ensure that the **placement restriction** on a BCP will not be violated.

Computing and Storing Future Barrier Sets

The following algorithm is used to associate the FBS set with each control point.

- Generate the control flow graph for the program. Techniques to generate the control flow graph (CFG) are well known (see [20], for example). Figure 3 (a) gives the CFG for the program in Figure 1.
- Generate the control point graph. The control point graph (CPG) is obtained from the control flow graph by identifying the node with control points and then linking them. A control point node n_i is linked to another control point node n_j if the path from n_i to n_j in the CFG does not contain any other control point. Figure 3 (b) gives the CPG computed from the CFG of the program in Figure 1.
- Compute and associate with each control point its future barrier set. This computation is described immediately following this algorithm.
- Some compiler techniques can be used to finally prune the CPG. For example, after unrolling the inner loop in the example of Figure 1, we get the CFG which is shown in Figure 3(c). The corresponding CPG is given in Figure 3(d) By using well known compiler techniques like constant propagation and dead code elimination, we can prune the CPG. For example, assuming that the condition *Odd(j)* in program shown in Figure 1 is compile time computable, the CPG of Figure 3(d) can be pruned to that shown in Figure 3(e) by noting that initial value of barrier instance variable j has the initial value of one. Observe that this pruning results in FBSs with smaller sizes (compare the FBSs in Figures 3(b) and 3(d)).

The Future Barrier Set for each barrier and each instance is computed in a hierarchical manner. The nodes of the control point graph are grouped together into supernodes. Each supernode has exactly one entry node and one exit node and all the paths from the entry node pass through the exit node, i. e. , the exit node post-dominates the entry node. Figure 4 (a) shows such a hierarchical

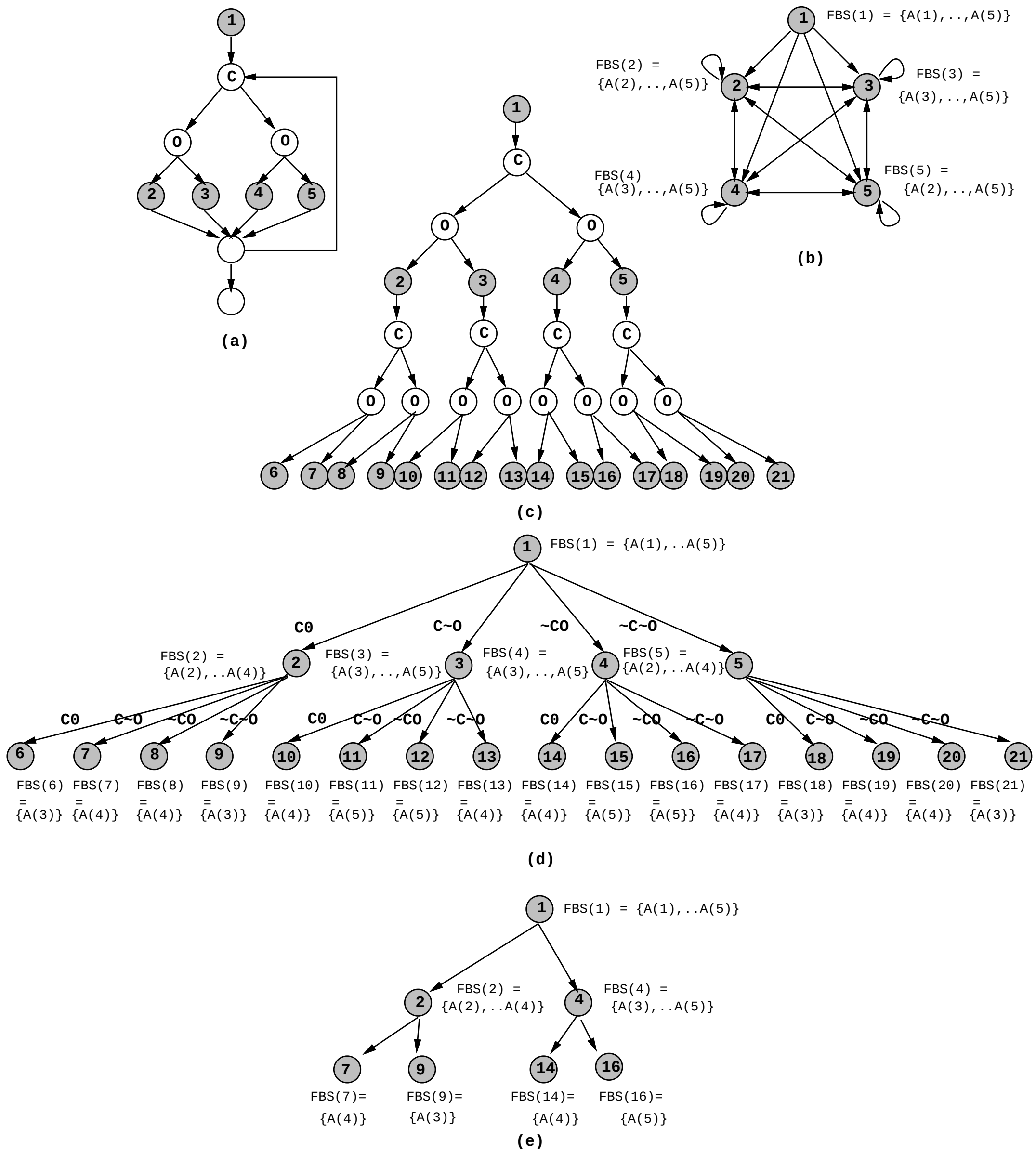

Figure 3: a) CFG for the example in Figure 1, b) CPG for CFG in (a), c) CFG with inner do loop unrolled , d) CPG for CFG in (c), e) CPG after pruning the unreachable nodes. The shaded nodes are the CP nodes. The nodes labeled 1,2,3,4 and 5 are the CP nodes corresponding to the BCPs in statements 3,8,9,14 and 15 of Figure 1, respectively. The label C is for nodes in which the condition *cond* is checked and the label O is for nodes in which the condition $Odd(j)$ is checked. The edges marked with $\mathbf{C}$,$\mathbf{0}$,$\tilde{\mathbf{C}}$ and $\tilde{\mathbf{0}}$ are the condition for taking that branch.

structure. Further, we assume that the CFG of the program is reducible[4], so that the entry node of a supernode will dominate all the nodes in that supernode. The computation of the future

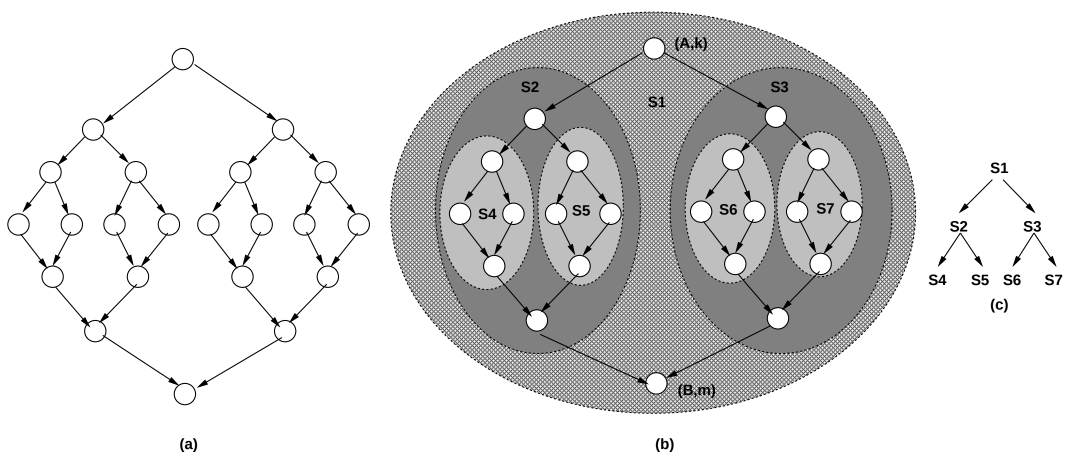

Figure 4: a) Control Point Graph b) Supernode grouping c) Hierarchical Supernode Structure

barrier set is further simplified by noting that

Theorem 4.1

$$FBS(entry\_point\_of(Supernode_j)) \;=\; FBS(exit\_point\_of(Supernode_j)) \bigcup_{n_i \in Supernode_j} (FBS(n_i))$$

where n_i is any node in $Supernode_j$ except its entry or exit node.

Proof: From the definition 3.2, it is obvious that FBS of a node n_k should be the superset of all the nodes reachable from it. Noting further that all the paths from the entry point of a supernode pass through the exit point yields the above equation.
□
The FBS of the entry point is the union of the FBSs of its children. Note that the FBS of a child node is no bigger than that of the parent node in the control point graph. If there is a loop in the control point graph then the FBS of the exit point is same as the FBS of the entry point of the loop. Assuming that the loop is a single-entry single-exit loop, the FBSs of all the control points in the loop is equal to the FBS of the entry point of the loop. The computation of the FBS can be formulated as a backward dataflow problem on the control point graph. As the FBSs of the nodes in a strongly connected component of the CPG are all nearly equal, the computation of the FBS can be efficiently done using the acyclic condensation of the CPG.

The above observation leads to the efficient storage of the FBS information for each control point in an incremental fashion. For example, the FBS of an entry node could consist of the storage

[4] A control flow graph G is reducible iff the edges of G can be partitioned into edge sets E_1 and E_2 such that the maximal acyclic partial flow graph has the edges E_1 and for every edge $(v_1, v_2) \in E_2$, v_2 dominates v_1 [20].

for the barriers in that supernode and a pointer to the FBS of the exit point of that supernode. By storing the FBS set in this incremental fashion, the space required for keeping all the FBSs can be reduced considerably. The hierarchical supernode structure also assists in eliminating the need to determine the process set of some barrier instance of a future barrier once the process set of some instance of a particular barrier is known. This observation leads to the following theorem:

Theorem 4.2 *The process set of the barrier associated with the exit point of a supernode is same as the process set of the barrier associated with its entry point.*

Proof: Obvious from the definition of a supernode.
□
For example, in Figure 4 (b), suppose the process set for instance k of barrier A, which is associated with the entry point of supernode S1, is known to be set P. The process set of instance m of barrier B, which is associated with the exit point of S1, is also P. This will considerably reduce the amount of computational overheads for determining the process set.

5 A Possible Implementation

We first present an architecture which can efficiently implement the GDGB mechanism. This architecture, similar to any large scale (logically) shared memory multiprocessor, consists of a set of n processors (each with its own local memory) and a set of m memory modules interconnected by a multistage network. In addition, it has a separate processor called the barrier processor which is connected by a bus to all the other processors. The barrier processor (BP) has its own local memory which we call the barrier memory (BM). The barrier processor continually monitors the processes' states and hence is granted read only access to the local memory of each processor where the processor maintains the process state of each process that is currently active on the processor. Since there is a one way communication from the barrier processor to each of the other processors, there is no bus contention. Building such a bus would also not be very expensive compared to the overall cost of the multiprocessor. From the information derived from each process' state, the barrier processor maintains various data structures (for example, the control point graph) in the barrier memory. The barrier processor also has access to the global memory in order to notify processes of the completion of the formation of a process set. The architecture suggested above could, of course, be simulated on traditional large scale multiprocessors with some performance degradation. Consider, for example, its simulation on an architecture such as the BBN Butterfly. A processor could be identified as a barrier processor, this processor's local memory would then be the barrier memory. The process state needs to be maintained in the global portion of each processor's memory. The performance degradation arises since the barrier processor generates global traffic to monitor the process' states.

For this implementation, the GDGB mechanism works as follows:

1. At compile time the control point graph is generated and stored in the barrier memory. The sure set and the probable set for each barrier is determined. If the compiler is unable to

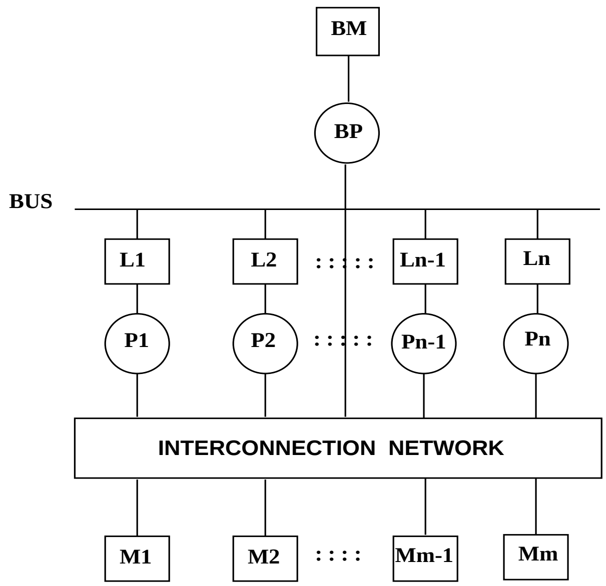

Figure 5: Barrier Architecture

determine whether or not a process will participate in a particular barrier, say (A, k), then the process is included in the probable set $PS(A, k)$.

2. At run time the barrier processor continually monitors the active control point[5] of the processes in the probable set of a barrier. The barrier processor uses the FBS information and the current instance of the barrier variable in deciding whether the process will be participating in a barrier. Associated with each control point there is a future barrier set. Until an active barrier instance, say (B, m), is in the future set of a process,say P, P is a possible candidate of the process set of (B, m). Hence a barrier is not complete until it is present in future barrier set of some process. As soon as the barrier processor is able to determine that a process, say Q, in the probable set $PS(B, M)$ will definitely participate in the barrier (B, m), Q is placed in the sure set $SS(B, m)$. The process is deleted from the probable set $PS(B, m)$. Hence, as the computation proceeds, the size of the probable set for each barrier keeps reducing. When the probable set of a barrier, say (A, k), becomes empty then the barrier processor sets a boolean $Barrier\_proceed(A, k)$ to $true$ indicating that the process set recognition phase for barrier (A, k) has completed. The barrier processor also determines the size of the process set (which is equal to the size of $SS(A, k)$) and records it. Once a particular process set is complete, processes can barrier synchronize.

3. For the basic barrier reporting and notification phase any known scheme which fits with the above mechanism can be used. For example, the accumulating barrier scheme can be modified to take care of the fact that the process set size is not known at compile time.

If the control point occurs much earlier than the barrier instruction itself, then the barrier processor can decide in advance whether or not a certain process will be participating in a barrier.

[5]The active control point is maintained as part of a process' state.

Measure	Adaptive	GDGB-Adaptive
Number of accesses per process for a Barrier instance	$O(log(\|SS\| + \|PS\|))$	$O(log(\|SS\| + \|PS\_S\|))$
Delay seen by q $\in PS\_U$	$O(log(\|SS\| + \|PS\|))$	$O(1)$

Table 1: Comparisons with other barrier scheme Barrier Schemes

Hence the compiler attempts to place a control point so that its execution occurs much earlier than the execution of the associated barrier instruction.

6 Performance Issues

Almost any barrier synchronization scheme, with suitable alterations, can be enhanced with the GDGB mechanism. To compare performance, then, we consider a particular barrier synchronization scheme with and without the enhancement. As an example, we use the adaptive combining tree based barrier synchronization scheme, which is a particularly efficient process based synchronization scheme. Table 1 compares an enhanced and the unenhanced versions of this scheme. The first measure compares a lower bound on the number of accesses for the completion of the barrier. This lower bound is not tight since the number of accesses grows superlineraly with the number of processes. For the adaptive scheme, all the processes in the set $(PS \cup SS)$ need to synchronize-hence, the number of accesses per process is $O(log(|SS| + |PS|))$. For the GDGB adaptive scheme, on the other hand, the number of accesses is $O(log(|SS| + |PS\_S|))$. If p is the probability that a process in the probable set meets at the barrier, then the number of accesses in the enhanced scheme can be estimated to be $O(log(|SS| + p * |PS|))$ which lead to considerable savings for large probable sets and small values of p. The second measure compares the delay seen by a process in the PS_U set. For non-GDGB mechanism, it is proportional to the number of accesses required by each process to synchronize which is $O(log(|SS| + |PS\_S|))$. The delay seen by the GDGB enhanced scheme is a constant, however. Finally consider the following scenario: all the processes except an uninterested one in the PS_U set have arrived at a barrier instance. Let t be the delay seen by these waiting processes for the uninterested process to barrier synchronize in the non-GDGB scheme. The delay seen with the GDGB enhancement is lesser and equals $(t - d)$ where d is the time to execute the program fragment between the proper BCP and the associated barrier instruction. Let P_g be the probability that the GDGB detects the completion of the barrier before a non-GDGB scheme. And let p be the probability that a process in the probable set participates in the barrier. Then assuming that all the arrival orders are equally probable, P_g is equal to the probability that the last process arriving is in the PS_U set of the barrier:

$$\begin{aligned} P_g &= \frac{|PS\_U|}{|PS| + |SS|} \qquad (2) \\ &= (1 - p) * |PS|/(|PS| + |SS|) \end{aligned}$$

We plot P_g against the ratio of the sizes of the probable and the sure sets for various values of p (Figure 6). The plot shows that there is a probability of greater than 0.5 that the last arriving process belongs to PS_U for smaller values of p ($0 < p \leq 0.4$) and when $|PS| \geq |SS|$. Of course, more analysis needs to be done to exactly quantify the unnecessary delay in the completion time because of the uninterested processes.

7 Examples

In this Section we present two examples where the GDGB scheme is more useful than other barrier schemes.

7.1 Parallel Balanced Merge Sort

We use the parallel balanced merge sort algorithm proposed by Francis and Mathieson [6] who present a parallel merge sort algorithm in which all the p processes are used at each of the $\log p$ merging stages. We present the structure of the algorithm in Figure 7, with GDGB calls, to illustrate how generalized dynamic group barrier can be used in real programs. Referring to Figure 7, barriers are needed initially, after partitioning, and after merging.

The barrier directed acyclic graph (dag) for the program in Figure 7 is shown in Figure 8. From the barrier dag one can see that the process set for the same barrier differs from instance to instance. For example, the process set for B0(1) is {P0,P1} whereas process set for B0(3) is {P0,..,P8}. Also it can be seen that the process set of B1(1) and B1(2) are disjoint and so the two different instances of the same barrier can execute in parallel.

7.2 Parallel Garbage Collection

Traditional parallel garbage collection (GC) algorithms adopt a processor based view and assume that all processing be stopped at a barrier before garbage collection can take place. Such an assumption is unrealistic in a multithreaded environment and hence traditional parallel GC schemes are not applicable [13]. Threads may be dynamically rescheduled, either because they block, or simply because they are processes in a time-sharing system. Therefore there is no reliable binding of a thread to a processor. In such an environment, acquiring a global barrier requires blocking all the running threads, and to wait until all the runnable threads have been scheduled and then blocked. It is difficult, however, to decide when the barrier has been acquired, because even if the total number of threads is known, it is not known how many threads are blocked and waiting for other conditions. Hence there is a need for a mechanism wherein only m out of n threads synchronize at the barrier, where m is determined dynamically. Different criteria can be chosen to determine the value of m, for example, the value of m needs to be determined when the pointers which point to free blocks of memory need to be a closed set [13]. Since m is determined dynamically, the GDGB scheme is the only appropriate barrier synchronization scheme.

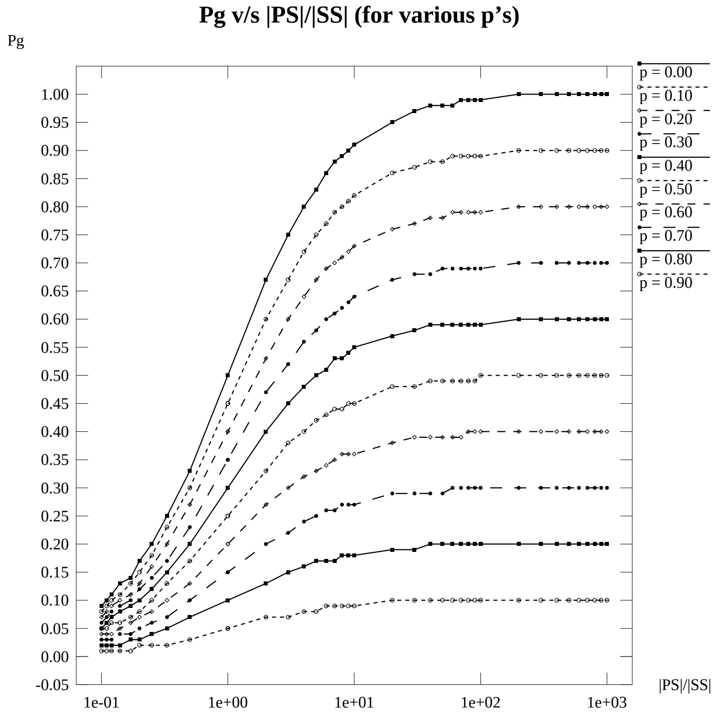

Figure 6: P_g v/s $|PS|/|SS|$ for various values of p

```
doall i = 0 to p-1;
    k=1;
    Sort n/p values;
    GDGB(⌊i/2⌋,k);
    k=k+1;
    do j = 1,log p;
        Partition(i,j);
        GDGB(⌊⌊i/2⌋/j⌋,k);
        k=k+1;
        Merge;
        GDGB(⌊⌊i/2⌋/j⌋,k);
        k=k+1;
    enddo ;
 enddoall ;
```

Figure 7: Parallel Balanced Merge Sort Algorithm

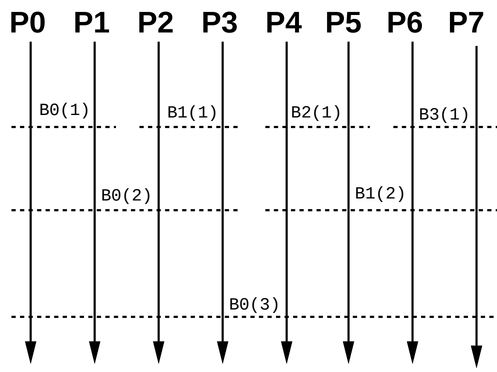

Figure 8: Barrier Dag

8 Summary and Conclusion

We expect the GDGB mechanism to be efficient because of the following reasons:

1. Smaller number of processes need to synchronize as only the interested processes are involved in the barrier synchronization. Consequently, one could expect lower overheads than other schemes since synchronization overheads usually grow superlinearly with the number of processes.

2. Since uninterested processes do not participate in the barrier synchronization , they continue their execution without any delays introduced by barrier synchronization.

3. The GDGB mechanism exploits the use of concurrent barriers at a finer level of granularity than all the other schemes proposed in literature. Since in this scheme two different instances of the same barrier can execute concurrently. In other schemes only two different barriers can execute concurrently. Stating this in "parallel loop terminology"[6], GDGB exploits concurrency both at the inter-loop and the intra-loop levels, whereas other mechanisms exploit concurrency only at the inter-loop level.

4. Process recognition phase and the barrier recognition phase can proceed in parallel.

To perform an accurate modeling of the GDGB mechanism, the contention in accessing the synchronization variables and the variance in the process arrival rates need to be taken into account. Our modeling, which is conservative and preliminary, shows that the GDGB mechanism performs well. We expect that the situation in the practice would be even better when dynamic sets of processes need to barrier synchronize. We plan to evaluate the performance of this scheme through simulation. A possible criticism of our scheme could be the large amount of information that needs to be generated at compile time. This may not be a problem considering the sophisticated processing that parallelizing and vectorizing compilers do. The analysis required for the GDGB scheme can be easily accommodated into the dataflow analysis which a compiler has to anyway

[6] Barriers are mainly used to synchronize the termination of iterations of a parallel loop

do. Also, since many of the barriers are introduced by the parallelizing compiler itself, the task of generating extra information for them would appear straightforward.

Acknowledgement

The authors would like to thank W. Kuchelin, J. Martens, and D. Panda for many insightful discussions on the subject.

References

[1] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing*, (3):129–140, 1986.

[2] C. J. Beckmann and C. D. Polychronopoulos. Fast barrier synchronization hardware. In *Supercomputing*, pages 180–189, 1990.

[3] Y. Birk, P. B. Gibbons, J. L. C. Sanz, and D. Soroker. A simple mechanism for efficient barrier synchronisation in MIMD machines. In *Proc. of International Conference of Parallel Processing*, volume 2, pages 195–198, Aug 1990.

[4] Alliant Computer Systems Corporation. Fx/series architecture manual. Technical Report Part Number 300-00001-B, Alliant Computer Systems Corporation, Jan. 1986.

[5] I. Dimitrovsky. ZLISP- a portable parallel LISP environment. Doctoral Dissertation, Courant Institute,New York,NY, 1988.

[6] R. H. Francis and I. D. Mathieson. A benchmark parallel sort for shared memory multiprocessors. In *IEEE Trans. on Computers.*, volume 37(12), pages 1619–1626, Dec. 1988.

[7] E. Freudenthal and A. Gottlieb. Process coordination and fetch-and-increment. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, pages 260–268, Apr. 1991.

[8] K. Ghose and D-C Cheng. Efficient synchronization scheme for large-scale shared memory multiprocessors. In *Proc. of International Conference of Parallel Processing*, volume I, pages 153–160, 1991.

[9] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAulffe, L. Rudolf, and M. Snir. The NYU Ultracomputer - designing a MIMD, shared-memory parallel machine (extended abstract). In *IEEE Trans. Computers*, pages 175–189, Feb. 1983.

[10] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

[11] K. Hwang and S. Shang. Wired-nor barrier synchronization for desigining large shared- memory multiprocessors. In *Proc. of International Conference of Parallel Processing*, volume I, pages 171–175, 1991.

[12] D. N. Jayasimha. Distributed synchronizers. In *Proc. of International Conference of Parallel Processing*, pages 23–27, Aug 1988.

[13] W. Kuchlein and N. J. Nevin. On multi-threaded list-processing and garbage collection. Technical Report TR11, Dept. of CIS,OSU, March 1991.

[14] B. Lubachevsky and A. Greenberg. Simple,efficient asynchronous parallel prefix algorithm. In *Proc. of International Conference of Parallel Processing*, pages 67–67, Aug. 1987.

[15] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Apr. 1991.

[16] M. T. O'Keefe and H. G. Dietz. Hardware barrier synchronization: Dynamic barrier MIMD(DBM). In *Proc. of International Conference of Parallel Processing*, volume 1, pages 43–46, Aug 1990.

[17] M. T. O'Keefe and H. G. Dietz. Hardware barrier synchronization: Static barrierMIMD(SBM). In *Proc. of International Conference of Parallel Processing*, volume 1, pages 35–42, Aug 1990.

[18] A. Osterhaug. Guide to parallel programming on sequent computer systems. Technical report, Sequent Technical Publication, Sequent Computer Systems,Beaverton,Oregon, 1987.

[19] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3). In *Proc. of International Conference of Parallel Processing*, pages 764–771, 1985.

[20] Zima and Chapman. *Supercompilers for Parallel and Vector Computers.* ACM press, 1 edition, 1990.