Capability Classes of Barrier Synchronization Techniques

R. DeMara[†], Y. Tseng[‡], K. Drake[†], A. Ejnioui[†]

[†] Department of Electrical and Computer Engineering
University of Central Florida
Orlando, Florida 32816-2450

[‡] Department of Computer & Information Sciences
Florida A&M University
Tallahassee, Florida 32307

***Abstract*—Performance metrics and evaluation criteria are used to develop a novel taxonomy which classifies barrier mechanisms into categories ranging from simple Static-Binding Idle-Tasking (SBIT) methods to robust Dynamic-Binding Any-Tasking (DBAT) methods. Such capabilities include support for multiple barriers, reconfigurable and reusable synchronization resources, and provisions for dynamic process binding. Based on these features the existing software-based approaches, dedicated-hardware mechanisms, and hybrid hardware/software techniques for synchronization are assessed in order of increasing complexity. Any barrier synchronization algorithm can be readily classified using this taxonomy to understand resource requirements and performance tradeoffs.**

I. INTRODUCTION

Efficient barrier synchronization and quiescence detection techniques are essential for optimizing throughput in multiple processor architectures. An ensemble of processing elements (PEs) is said to be synchronized, or to have reached a quiescent state [1], upon completion of each interval of concurrent activity. Points at which synchronization occur are referred to as barriers [2-4]. The design objective is to minimize the overhead required to enforce completion of each barrier prior to the resumption of subsequent processing. Although the survey in [5] presents different criteria for classifying termination detection algorithms in distributed networks by abstracting away the hardware and software resources, this survey on the other hand presents a new set of criteria for classifying barrier synchronization and termination detection algorithms by considering the specifics of these resources in parallel and distributed systems.

A. The Barrier Synchronization Problem

Figure 1 shows a code fragment containing three statements, labeled S_1, S_2, and S_3, which invoke three distinct processes labeled P_1, P_2, and P_3. Let $I(S)$ and $O(S)$ denote the set of input and output variables respectively of statement S. Statement S_1 and S_2 have no input-output nor control dependencies, and hence by Bernstein's conditions [6]:

$$I(S_1) \cap O(S_2) = I(S_2) \cap O(S_1) = O(S_1) \cap O(S_2) = \varnothing$$

An empty intersection implies that S_1 and S_2 can be executed simultaneously on separate processors. On the other hand, statement S_3 can only be executed correctly after both S_1 and S_2 have terminated since $I(S_3) \cap O(S_1) = \{x\}$ and $I(S_3) \cap O(S_2) = \{y\}$. The barrier, which corresponds to the completion of the concurrent processing, must occur before S_3 is initiated. This is indicated by the coend statement shown in Figure 1. The processing tasks between barriers are executed by multiple PEs of a parallel or distributed system. These PEs may execute these tasks simultaneously without impacting correctness granted that the barriers are properly enforced. Since some barriers may only involve a subset of the processes or resources in the system, those which actually take part in a specific barrier are delineated as *participating tasks* or *participating PEs* accordingly.

B. Application-Driven Synchronization Requirements

The selection of a barrier mechanism can be determined based on specific characteristics such as the application's task granularity between barriers, number of simultaneous barriers, and task creation/allocation strategy.

```
      cobegin;
S1:         x := P1(a);
S2:         y := P2(b);
      coend;    ←—— "Barrier" (point at which interprocess synchronization must occur)
S3:   P3(x, y);
```

Figure 1. Parallelizable code fragment requiring synchronization.

1) Granularity of the Application Tasks

Task granularity refers to the number and relative complexity of the operations within each concurrent process. The coarseness or fineness of granularity determines the interval of productive execution between barriers. As the tasks requiring synchronization become increasingly fine-grained, the relative impact of synchronization overhead on processing throughput becomes magnified.

2) Degree of Thread Concurrency

Singly-threaded applications require at most one barrier at any instant while *multi-threaded* applications may take advantage of concurrent barriers that are simultaneously active. For example, in a multi-user environment, each user job involves tasks that push code execution towards distinct barriers. Since there are no data dependencies between tasks from different users, these tasks could be executed simultaneously if the synchronization mechanism could distinguish between barrier signals.

3) Apriori Knowledge of PE Participation

Knowledge of whether a PE will participate in a barrier may not be readily available at compile-time. Applications in which the number of participating tasks and/or their processor binding can be determined prior to execution are said to exhibit *procedural task creation*. On the other hand, applications which dynamically select the PEs which will participate in the barrier and/or generate new processes based on run-time conditions are capable of *adaptive task creation*. Applications requiring synchronization support for adaptive process creation include Remote Procedure Calls, recursive algorithms, and dynamic search strategies.

C. Architecture-Driven Synchronization Requirements

Each phase of the synchronization algorithm must accommodate the primary architectural features of the target machine such as its communication mechanism, level of hardware support for synchronization, and various machine-specific parameters.

1) Interprocessor Communication Strategy

Since shared-memory architectures provide a common region of the address space which can be accessed by multiple PEs, applicable barrier techniques involve the use of *global synchronization variables*. The design objectives involve minimizing contention for access to these variables. On the other hand, distributed memory machines must exchange *synchronization messages* through the machine's interconnection network. Thus, design objectives for distributed-memory synchronization schemes involve minimizing message traffic, transit times, and computational overhead required to process these messages.

2) Machine-Specific Configuration Parameters

Regardless of whether a shared or distributed memory model is used, quantities such as the ratio of computation-to-communication speed can be determining factors in the applicability of a barrier technique. For instance, a synchronization algorithm may be applicable to a distributed-memory architecture, but the relative cost of communication on a particular machine may make certain approaches intractable. Similarly, the number of PEs in the machine, PE interrupt support, and spinlock availability will influence which barrier approaches are appropriate.

3) Availability of Barrier Hardware

The use of dedicated hardware to enforce barriers can significantly reduce synchronization latencies. The hardware design issues involve minimizing the logic requirements per PE and reducing interprocessor wiring complexity while optimizing flexibility. Use of dedicated barrier hardware can be prohibitive since many systems offer little hardware support. In addition, retrofitting these systems may sacrifice the application's portability. However, various vendors began to incorporate lately hardware support for barrier synchronization in order to realize further decreasing ratio of computation-to-communication speed by exploring new interconnect technologies. For example, the adoption of the SOME bus architecture based on optical bus technology can reduce significantly this ratio in new computing systems [7].

D. Capability Category

Given existing barrier synchronization mechanisms, we propose the novel categorization shown in Figure 2 based on the features of these mechanisms. These mechanisms are grossly classified as *static-binding* and *dynamic-binding* according to the way in which they allocate PEs and schedule processes. Only the mechanisms which can both allocate PEs and schedule processes dynamically are categorized as dynamic-binding. Although some approaches schedule processes dynamically, they fulfill the tasks on fixed trees of PEs, and subsequently they are also classified as static-binding. The barrier synchronization mechanisms are further classified as *idle-tasking, same-tasking, different-tasking*, and *any-tasking* under each binding scheme based on how they behave after they enter the barrier:

(i) If the joining PEs cannot be reactivated for other tasks after they enter the barrier, the mechanism is classified as idle-tasking capable.

(ii) If the joining PEs can be reactivated for other tasks in the same barrier after they enter the barrier, the mechanism is classified as same-tasking capable.

(iii) If joining PEs can only be reactivated for other tasks in other barriers after they enter the barrier, the mechanism is classified as different-tasking capable.

(iv) If the joining PEs can be reactivated for other tasks in either the same barrier or other barrier after they enter the barrier, the mechanism is classified as any-tasking capable.

Combined with binding classification, there are eight categories for barrier synchronization mechanisms.

E. Class Hierarchy of Barrier Techniques

Figure 3 shows a hierarchy of capabilities for the barrier classes defined in the previous section. In particular, class A is said to *subsume* class B if the mechanisms in class A can perform the operations of those in class B. For example, a technique in the DBAT class can correctly execute any synchronization operation supported by any other class. Therefore, the DBAT class subsumes all other classes. If one can realize DBAT class capability with a cost and efficiency comparable to any other class, it will provide a general technique for the barrier synchronization problem. Each of these classes is used below to compare and contrast the various barrier techniques available.

II. EVALUATION CRITERIA FOR BARRIER MECHANISMS

The evaluation criteria described below are applicable to dedicated-hardware mechanisms, software-based approaches, and hybrid hardware/software techniques. The criteria include desirable barrier-handling features and fundamental performance metrics.

A. Functionality and Design Features

Although all correct solutions to the synchronization problem provide a means to enforce barriers, certain techniques realize more flexible use of the synchronization resources and/or the available processing capacity of the PEs in the machine as shown in Table 1.

B. Performance Metrics

Metrics for quantifying the performance of barrier mechanisms assess the time and space costs of enforcing barriers within an application.

1) Detection, Restart, and Synchronization Overheads

The *detection latency*, denoted by t_d, of a barrier mechanism refers to the time delay required to recognize the arrival of the last PE at the barrier following a burst of concurrent processing activity of duration t_p. Once the barrier is detected, there may be an additional PE *restart latency*, denoted by t_r, which corresponds to the delay incurred from when the barrier is detected to when the PEs resumes processing. Since the PEs participating in the barrier are idle, the total synchronization overhead time, denoted by $t_{so} = t_d + t_r$, should be minimized. The degradation in throughput is further compounded by the potentially large number of processors that are idle. Figure 4 illustrates the effect that synchronization latency and restart delay have on processor throughput while Figure 5 shows the effect of synchronization overhead on lost throughput for a 100 μsec barrier overhead. In the bar chart of Figure 5, the abscissa shows MIPS per PE while the ordinate shows lost MIPS due to the synchronization overhead. The chart shows the effect of lost MIPS based on the number of PEs and their performance levels.

2) Barrier Reconfiguration and Initialization Delay

Some barrier mechanisms require an initialization step to indicate the commencement of detection for an upcoming barrier.

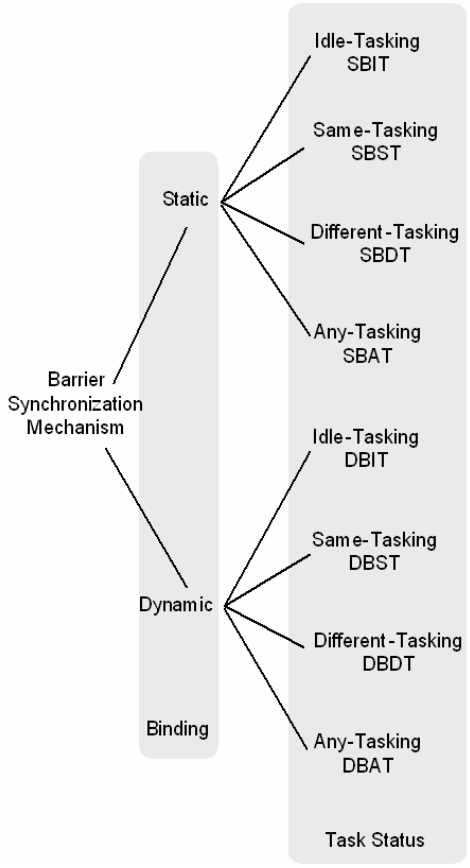

Figure 2. Classification scheme based on the functionality of the barrier.

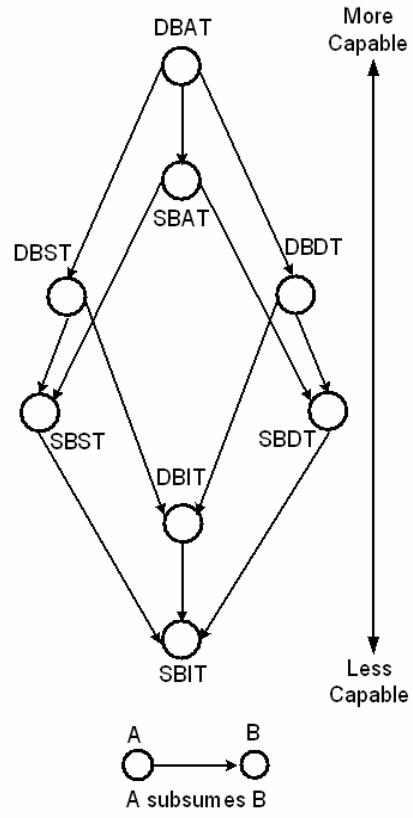

Figure 3. Hierarchy of barrier classes.

Capability/Limitation	Characteristics
Commitment vs. multiple-signaling of synchronization resources	In the presence of concurrent threads, some barrier techniques allow *multiple-signaling* while others require *commitment* of synchronization resources to a particular barrier. Commitment: When a PE signals that it has reached the barrier, a synchronization resource is said to be committed to that barrier if its further use is precluded until all other PEs also reach the barrier. Once committed, separate instances of these resources are necessary to distinguish among the contributing tasks and/or their parent barriers. The number of committed signaling resources is the product of the number of active barriers and the maximum number of participating PEs per barrier. Multiple-signaling: If each PE supports this signaling, the total resource instances can be reduced to the maximum number of PEs participating in any barrier.
Processor reactivation	*Reactivated PEs* perform a context switch while waiting for the remaining participants to arrive at a barrier. Although processor reactivation does not reduce the detection latency, it does reduce throughput loss due to stalled PEs.
Interconnection network	While some barrier synchronization mechanisms are designed for specific interconnection networks, other mechanisms are designed independently of network architecture thus making them implementable on any network without any limitation.

Table 1. Summary of resources and capabilities used in barrier synchronization.

Synchronization Latency

waste = (number of PEs) * (MIPS per PE) * t_{sync} * (barriers / sec)

where: $$\text{barriers / sec} = \frac{1}{\left\{ \frac{\overline{I_{barrier}}}{\text{MIPS per PE}} + t_{sync} \right\}}$$

$\overline{I_{barrier}}$: average number of instructions executed by a PE prior to the barrier used, which is 1 x 10^6.

Restart Delay

waste = N * (MIPS per PE) * t_{so} * (barriers / sec)

where: $$\text{barriers / sec} = \frac{1}{t_p + t_{so}}$$

Assumption: t_{so} is independent of MIPS.
Note: t_p is inversely related to MIPS, therefore waste is a function of the square of MIPS per PE. Waste is linearly related to t_p.

Figure 4. The effect of synchronization latency and restart delay on processor throughput.

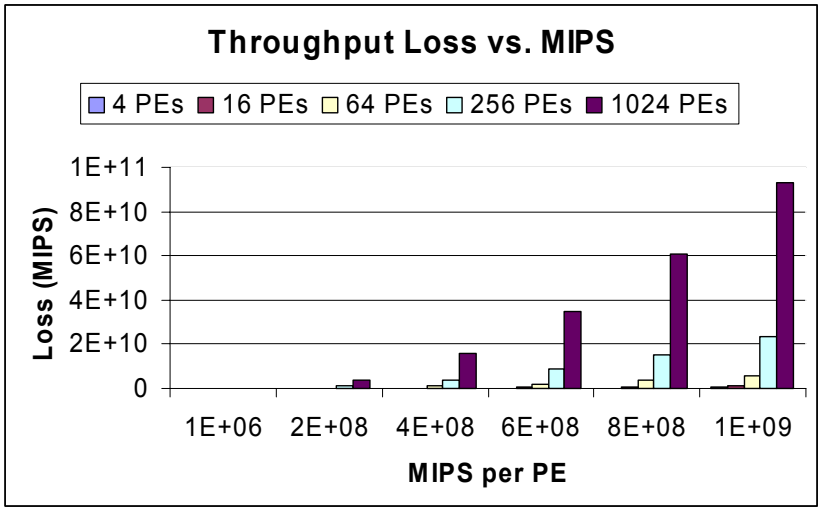

Figure 5. Throughput loss for a 100 μsec synchronization overhead.

Barrier reconfiguration may involve clearing software counters, setting hardware latches, or flushing message buffers. If not overlapped with PE computation, the *initialization delay* from these resetting operations comprises an overhead which is distinct from the barrier detection and PE restart latencies.

3) Degradation in Communication Capacity

The allocation of one or more communication resources to the synchronization mechanism causes a *degradation in communication capacity* which might otherwise be available for interprocessor communication of data values required for processing.

4) Degree of Processor-Oriented Scalability

Processor-oriented scalability refers to the feasibility and performance of the barrier mechanism as the number of processors in the machine is increased. Barrier mechanisms may exhibit varying degrees of processor-oriented scalability with respect to performance and/or cost. Scalable performance implies

that the synchronization latencies are well-behaved as the number of PEs is increased.

5) Degree of Thread-Oriented Scalability

To support a multi-user environment and single-user applications with concurrent and/or overlapped tasks, the synchronization method needs to be able to distinguish between multiple barriers. The *thread-oriented scalability* of a synchronization mechanism refers to its support for concurrent barriers as the number of these threads is increased.

III. STATIC-BINDING BARRIER TECHNIQUES

A. Static-Binding Idle-Tasking (SBIT) Capable Techniques

1) The Butterfly Barrier

The Butterfly Barrier is a barrier synchronization approach that is free of host spots and can incur a delay that grows logarithmically with the number of processors [2]. This technique builds upon a two-processor synchronization kernel illustrated in Figure 6. Statement S_1 guarantees that each processor will not continue to S_2 until the other processor has completed S_4 from the previous barrier. This prevents a race condition which can occur in the presence of short code segments or with processors that are subject to program interruption. S_2 signals entry of the barrier code to the other processor. In S_3, the processor waits until S_2 has been executed by the other processor. Finally, S_4 is used to re-initialize the flags f_0 and f_1 for the next barrier. In [2], the author proposed using this two-processor Butterfly lock to synchronize three or more processors using the structure shown in Figure 7. Multiple instances of the two-processor lock are employed to prevent any processor from proceeding beyond the barrier until all processors have reached the barrier. This barrier detection method is classified as SBIT. The technique is static-binding because it must embed the barrier synchronization code in each process in the sense that each process has to be known in advance. It is classified as idle-tasking since all PEs reaching the barrier code shown in Figure 6 must wait at S_3 for the period PE to execute S_2.

2) The U-cube Tree Algorithm

The U-cube Tree Algorithm [8] is designed for the wormhole-routed hypercube multicomputers by taking advantage of the feature that message latency is almost insensitive to the distance between the source and destination nodes in wormhole routing. Therefore it may not be efficient if implemented on other interconnection networks.

P1	**P2**
S1: while ($f0 \neq 0$);	S1: while ($f1 \neq 0$);
S2: $f0 = 1$;	S2: $f1 = 1$;
S3: while ($f1 \neq 1$);	S3: while ($f0 \neq 1$);
S4: $f1 = 1$;	S4: $f0 = 0$;

Figure 6. Two-process butterfly barrier.

The algorithm uses a barrier processor to do termination detection, which can be a joining PE or a dedicated processor. The algorithm takes part in (i) the distribution phase in which the barrier processor either broadcast or multicast the message to all the joining PEs, and (ii) the reduction phase in which the joining PEs report to the barrier processor for termination detection. The algorithm first organizes the joining PEs as a dimension-ordered chain which will be explained shortly. In the distribution phase, the barrier processor starts by unicasting to one of the joining PEs. Next, every PE, which has received a message, unicasts the message to one of the PEs, which have not received messages yet in the following steps, until all joining PEs receive a message. This phase requires exactly $k = \lceil \lg(m+1) \rceil$ steps. The reason for using dimension-ordered chain is to guarantee that the paths followed by concurrent messages in the U-cube tree do not go through any common channel. The dimension-ordered chain is formed by the three following definitions [8]. Let $\sigma_{n-1}(x)\sigma_{n-2}(x)\ldots\sigma_0(x)$ represent the binary address of a node.

Definition 1: The binary relation ``dimension order,'' denoted $<_d$, is defined between two nodes x and y as follows: $x <_d y$ if and only if either $x = y$ or there exists a j such that $\sigma_j(x) < \sigma_j(y)$ and $\sigma_i(x) = \sigma_i(y)$ for all i, $0 \le i \le 1$.

Definition 2: A sequence $\{d_1, d_2, d_3, \ldots, d_m\}$ is a dimension-ordered chain if and only if all the elements are distinct and the sequence is dimension-ordered, that is, if $d_i <_d d_j$ for all i, j, such that $1 \le i \le j \le m$.

Definition 3: A sequence $\{d_1, d_2, d_3, \ldots, d_m\}$ is called a d_0-relative dimension-ordered chain if and only if $\{d_0 \oplus d_1, d_0 \oplus d_2, \ldots, d_0 \oplus d_m\}$, is a dimension-ordered chain.

The U-cube Tree Algorithm is shown in Figure 8. An example based on (11010)-relative dimension-ordered sequence {01110, 01000, 11100, 11011, 00001, 01101} is given in Figure 9.

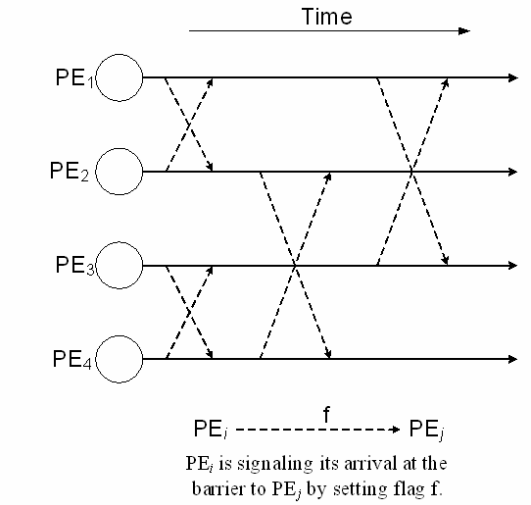

Figure 7. Butterfly barrier expanded to support multiple processors.

Algorithm: The U-cube Tree Algorithm
Input: d_0-relative cube ordered address $\{d_{left}, d_{left+1}, \ldots, d_{right}\}$
where d_{left} is the local address.
Output: Send $\lceil \lg(right - left + 1) \rceil$ messages
Procedure:
begin
$p = \lceil \lg(right - left + 1) \rceil$ messages
while (p > 0) do
$center = left + \left\lceil \frac{right - left + 1}{2} \right\rceil$;
$D = \{d_{center}, d_{centert+1}, \ldots, d_{right}\}$;
Send a message to node d_{center} with the address field D;
$right = center - 1$;
$p = p - 1$;
endwhile
end;

Figure 8. U-cube tree algorithm[8].

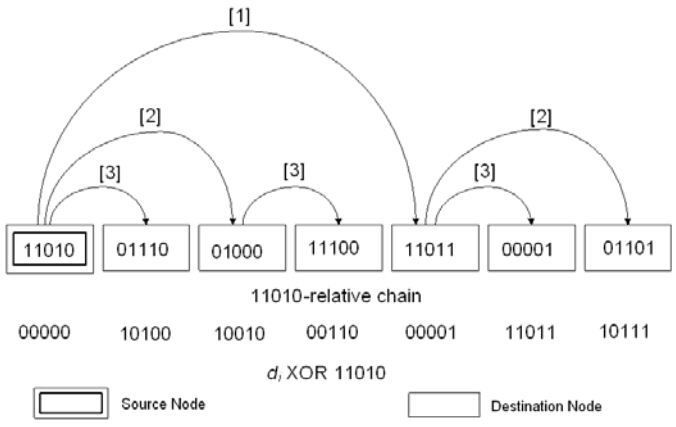

Figure 9. Example for the U-cube tree algorithm.

In the reduction phase, the algorithm just uses a reverse U-cube tree. This algorithm is classified as SBIT. It is static-binding because it needs to know the joining PEs to arrange the dimension-ordered sequence before it starts. It is idle-tasking since all PEs have to wait for the barrier processor for new messages.

B. Static-Binding Same-Tasking (SBST) Capable Techniques

1) The CV Algorithm

The procedure used in the CV algorithm [9] is shown in Figure 10. To begin, the CV algorithm organizes all the participating processors as a logical spanning tree of PEs. It can start after the underlying computation starts. When the CV algorithm starts, it flushes every link in the spanning tree to take care of messages sent before the algorithm has started. Next, the root node changes its state to DT (detecting termination), sends a *warning* message to each of its children, and color the link at the same time. In turn, the warning messages are passed through links connected to children nodes until all the participating nodes are notified of the detecting termination decision. The CV algorithm maintains a stack in each PE to keep track of sending and received activities on each PE. When a node becomes idle, it examines its stack from the top. For every received entry, it sends a *remove_entry* message to the sender and erases the entry from its stack. It repeats this until it encounters a sending entry. This procedure is defined as *stack_cleanup*. When a node receives a remove_entry message, it scans its stack and deletes the sending entry related to this message and repeats the stack_cleanup procedure as previously described if it is in idle status. A node sends a *terminate* message to its parent when it is idle, its stack is empty, each of its incoming links is colored, and it has received the terminate message from each of its children. When the root node meets the requirements to report the terminate message, it declares the termination. The CV algorithm is classified as SBST. It is treated as static-binding because it is performed on a fixed spanning tree of PEs formed before its execution. It is also classified as same-tasking since it supports only processor reactivation for the same barrier as originally defined.

2) The LTD Algorithm

The LTD algorithm [10] is an improvement over the CV algorithm. The algorithm for all PEs is shown in Figure 11. Like the CV algorithm, the LTD algorithm organizes participating processors as a spanning tree of PEs first. It start only after the underlying computation starts. When the root node decides to start the algorithm, it changes to DT status and sends a *start* message to each of its children. In turn, its children send start messages to their own children until all participating processors are notified of the root's decision. Each PE_i maintains four variables, namely in_i, out_i, $mode_i$, and $parent_i$, to apply *message counting* in order to decide whether all the messages sent by the same PE have been finished. The integer array, $in_i[1..n]$, is used to keep track of the messages that are received from PE_1 to PE_n and have not been finished. The number of messages sent by PE_j to PE_i is stored in $in_i[j]$. The integer, out_i, records the number of unfinished messages sent by PE_i itself. The Boolean variable, $mode_i$, shows the status of the processor (DT or NDT). The pointer, which indicates from where the most recent major message came, is stored in $parent_i$. A major message is a message received when the processor is idle and has finished all the messages that it sent to other processors. Otherwise the received message is defined as a minor message. Whenever a node turns idle, it calls procedures *respond_minor* and *respond_major* to detect termination. Procedure *respond_minor* sends one *FINISH*(k) message to each non-parent node to inform it of the number of messages it has finished for it. k represents the total number of messages which the procedure has finished for a specific node before turning idle. The LTD algorithm uses one *FINISH*(k) message instead of k remove_entry messages as in the CV algorithm to save ($k - 1$) messages. This saving represents a significant improvement in overhead minimization over the CV algorithm. On the other hand, procedure *respond_major* checks whether all the sent messages have been finished. Next, it sends one *FINISH*(k) message to the parent node after all the messages it has sent are finished. After the root turns idle and finds out that all the messages it sent out are finished, it concludes the termination. The LTD algorithm is classified as SBST based on the same rationale used to classify the CV algorithm.

C. Static-Binding Different-Tasking (SBDT) Capable Techniques

1) Collective Synchronization Tree

The Collective Synchronization (CS) Tree algorithm [11] is designed to run on 2D mesh networks. A CS tree is built on joining PEs before the algorithm begins. The CS tree is a logic tree which is rooted at the central node of the joining PEs and links all member nodes together. In short, a CS tree is built by (i) dividing the joining PEs into four quadrants according to their positions in the 2D mesh network, and (ii) assigning the central node as the root node. Next, it initializes the routers to set up the CS tree in hardware.

```
Procedure start;
(* performed by the root node when it decides to detect termination of the
   underlying computation *)

begin
       mystate ← DT;
       for each outgoing network link ln do
              begin
                     Color ln;
                     Send a warning message on ln;
              end
end;

Procedure receive_warning;
(* performed when a node p receives a warning message from its neighbor q *)

begin
       Color the incoming link (q, p);
       if (mystate <> DT) then
              begin
                     mystate ← DT;
                     for each outgoing network link ln do
                            begin
                                   Color ln;
                                   Send a warning message on ln;
                            end
              end
end;
```

Figure 10. Procedures used in the CV algorithm [9].

```
A1: (Upon sending a basic message to p_j)
       out_i := out_i + 1;

A2: (Upon receiving a basic message from p_j)
       in_i[j] := in_i[j] + 1;
       if (parent_i = NULL) and (i ≠ 1) then
              parent_i := j;

A3: (Upon deciding to switch to DT mode)      /* for p_1 */
        Or (Upon receiving a START message)   /* for p_i, 2 ≤ i ≤ n */
       mode_i := DT;
       for each child p_j of p_i do
              Send a START message to p_j;
       end for

       if (p_i is idle) then
              Call respond_minor(i);
              Call respond_major(i);
       end if

A4: (Upon receiving a FINISHED(k) from p_j)
       out_i := out_i - k;
       if (mode_i = DT) and (p_i is idle) then
              Call respond_major(i);

A5: (Upon turning idle)
       if (mode_i = DT) then
              Call respond_minor(i);
              Call respond_major(i);
       end if
```

Figure 11. Algorithm for p_i, $1 \le i \le n$, in the LTD algorithm [10].

To record the parent-child relationship in the CS tree, the routers use two sets of centralized registers, namely *status* and *working* registers [11]. Each status register contains two fields, *parent* and *child*, where each field has four bits (+X, -X, +Y, -Y). A "1" in any bit in a field indicates that the parent or child node can be reached through the corresponding port. In addition to these two fields, the status register contains a third field, called *node type*, which indicates the role of the node in the CS tree. A node in the CS tree can play the role of a central node, a leaf node, an internal node, or an intermediate node. The working status field in the working register set is used to record whether the message from the local processor (P field) or child nodes (child field) has arrived. Both registers for the same barrier are identified by a field called *Group ID*. Hence, the CS Tree algorithm can support the synchronization of different barriers simultaneously by applying different group ID. The operations of all nodes in the algorithm are summarized in Table 2.

In general, the leaf nodes report to their parents after they finish their tasks while the internal and intermediate nodes wait for messages from all their children and the local processor before they report to their respective parents. At the end, the central node declares the completion of the barrier after it has received messages from all its children. The CS Tree algorithm is classified as SBDT. It is static-binding because it must know the joining PEs a-priori to build the CS Tree. In addition, it has different-tasking capability since it employs different register sets for different barriers. An improvement to the performance of the CS tree is the *Barrier Tree for Meshes* (*BTM*) proposed in [12]. The BTM is a 4-ary tree whose height is smaller than the CS tree and subsequently results in lower routing latency. Contrary to the CS tree, nonmember nodes, which are nodes not involved in the barrier synchronization, are not included in the construction of the BTM tree. This exclusion of nonmember nodes reduces the setup overhead of synchronization and reduces significantly the synchronization latency. While the BTM improves the CS tree on 2D mesh networks, the *Barrier Synchronization* (*BS*) *Tree* presented in [13] improves the performance of barrier synchronization on irregular network topologies frequently used in cluster and distributed shared memory systems. A distributed algorithm constructs the BS tree when a barrier operation is performed for the first time. In subsequent barrier operations, messages can be delivered adaptively according to the hierarchy of the established tree to avoid congestion and possible faulty nodes in the interconnection network. Both synchronization approaches in [12, 13] fall in the SBDT category.

2) Fetch-and-Add

The *Fetch-and-Add* (F & A) primitive [6] is a hardware feature that allows a PE to indivisibly read and increment a counting variable stored in shared memory. If the number of processes converging on a barrier is known a-priori, a counting variable can be used to detect the barrier. An example of the code executing on the converging processes is shown in Figure 12. As each processor reaches the point in its operation where synchronization is required, it increments and tests the counting variable using an F & A instruction. When a process detects that the counting variable has reached the expected final value, the barrier has been reached. Both *counter* and *exit_flag* are initialized to zero prior to executing the barrier code. Note that the process which detects the barrier, that is the last converging process to reach the barrier, is allowed to execute sequential code since the *else* portion of the *if* statement can be executed by a process only if all other converging processes are in a busy-wait condition while testing the exit_flag. The use of this primitive for barrier synchronization can result in significant hot spots due to contention for both the counting variable and the exit_flag. The detection latency encountered when using the F & A primitive is determined by the access time of the counting variable and the test for the terminal count by the last PE reaching the barrier. It is clear that this synchronization method is SBIT. It is static-binding since joining PEs must be known in advance. In addition, it is idle-tasking since, with the exception of the last PE, all PEs reaching the barrier code must wait for the exit_flag to be set as shown in Figure 12. However concurrent barriers can be accommodated by duplicating the barrier code on multiple sets of PEs where each set is using a different counting variable. Thus a PE can be reactivated for a different barrier. In all, the F & A primitive can be classified as SBDT to show that most SBIT techniques can be easily upgraded to SBDT techniques.

D. Static-Binding Any-Tasking (SBAT) Capable Techniques

So far, we have not encountered any SBAT mechanism that was previously published in the research literature. However, some mechanisms in other categories can easily be adapted to the SBAT capability. For example, both the CV and LTD algorithms can be extended to be SBAT capable by attaching a barrier ID to each message in order to help multiple barriers to execute simultaneously without ambiguity.

Node Type	Operations
Leaf	**RA1**: Receive a Rmsg from local processor and forward it through the port specified in SReg[PF].
Internal	**RB1**: Receive a Rmsg (possibly from local processor) and set the bit corresponding to its input port in WReg. If SReg[CF] ≠ WReg[CF] or WReg[P] is not set, discard the message and go to RB1.
	RB2: Forward the message through the port specified in SReg[PF].
	RB3: Reset WReg.
Central	**RC1**: Same as RB1.
	RC2: Reset WReg and notify local processor.
Intermediate	**RD1**: Receive a Rmsg and set the bit corresponding to its input port in WReg[CF]. If SReg[CF] ≠ WReg[CF], discard the message and go to RD1.
	RD2: Same as RB2.
	RD3: Reset WReg.

Table 2. Operations in the router for the CS tree [11].

```
num_at_barrier = F&A(counter, 1);
if (num_barrier < num_expected)
       while (exit_flag == 0);
else
       {sequential code};
       exit_flag = 1;
end if;
```

Figure 12. Fetch and Add barrier code.

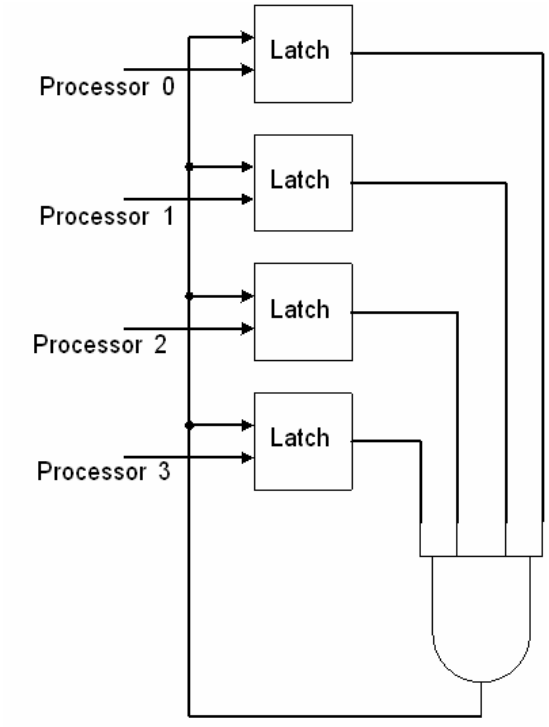

Figure 13. Simple AND gate barrier.

IV. DYNAMIC-BINDING BARRIER TECHNIQUES

A. Dynamic-Binding Idle-Tasking (DBIT) Capable Techniques

1) AND Gate Barrier

Ghose and Cheng propose a simple AND gate hardware barrier shown in Figure 13 [14].Each processor declares its arrival at the barrier by setting a local latch. This figure shows a five-processor synchronization circuit. The block containing the symbol is a latch that can be set by the processor when the latter reaches the barrier. The outputs of all the latches are AND-ed together, thus generating a global reset signal to all latches. It is classified as DBIT. It is dynamic-binding because it does not need to know the participating PEs in advance. Also, it is idle-tasking because every joining PE has to wait for the global reset signal after that PE has reached the barrier.

2) TTL_PAPERS

The TTL_PAPERS [15] is a simple TTL hardware implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization (PAPERS) [16]. It is plugged into the parallel ports of the personal computers connected to the cluster. Conceptually, the TTL_PAPERS employs an AND tree to detect whether every PE has reached the barrier. However there are two serious problems with AND trees in asynchronous barriers: (i) A PE with a small task for the first barrier may reset its signal before all other PEs signal for completion of the first barrier. In this case, the PEs may end up waiting for a signal that has been de-asserted. (ii) A PE which finishes the first barrier faster may assert the signal again before other PEs clear their signals for the first barrier. The TTL_PAPERS adopts the two NAND trees and the one-bit register design shown in Figure 14 to handle problem (i) and (ii). To solve problem (i), it connects the NAND trees to a one-bit register. When all PEs signal completion of a barrier, the output of the first NAND tree sets the register to 1, after which every PE can test the RDY signal to know whether the barrier has been reached. To solve problem (ii), it incorporates a second NAND tree to the hardware implementation. Any PE sets signal *S*0 after it clears the signal *S*1. When all PEs set their *S*0 signal, the output of the second NAND tree resets the register. Any PE can enter the next barrier after the RDY signal is reset. The TTL_PAPERS approach is classified as DBIT. It is dynamic-binding since it does not need to know the joining PEs. However since every PE has to be committed to the barrier, it is idle-tasking capable.

B. Dynamic-Binding Same-Tasking (DBST) Capability

1) Simultaneous Access Variable

The *Simultaneous Access Variable* (SAV) [17] technique is a simultaneous access design which provides idle processor reactivation and termination detection capabilities for the shared-memory architecture. The SAV algorithm organizes PEs as a binary tree and uses a shared queue to store spawned tasks. The values sent to the SAV by every PE under different situations are listed in Table 3. Every PE keeps track of the accumulated SAV of its subtree and reports it to its parent. The SAV acts like a counter that counts the difference of the idle PEs and the tasks inserted into the shared job queue. If the value of SAV is greater than zero, there are more idle PEs than the tasks inserted to the queue. Otherwise, there are more spawned tasks than available idle PEs. The termination is reached when the accumulated SAV at the root equals the number of the PEs in the system. The PEs at odd and even levels of the tree execute alternately. Figure 15 shows the algorithm for each active PE. Each PE maintains four sets of registers, each consisting of *R* and *U* registers. Three sets of registers are for information received from or intended for the left child, the right child, and parent respectively. These registers are denoted by the subscripts of *l*, *r*, and *p* respectively. The remaining register set is used for other tasks. When a left child PE sends *R* to its parent, the parent stores it in R_l; otherwise it is stored in R_r. On the other hand, the value received from the parent node is stored in U_p. The value of U_p, $U_p \geq 0$, is the number of tasks sent by the parent of the node. These tasks may be consumed in the node's subtree. Every PE behaves based on the values of R_l and R_r. When $R_l > 0$ and $R_r > 0$, which means both child subtrees have more idle PEs than spawned tasks, tasks from the PE's parent are shared among two children proportionally. If $R < U_p$, only *R* of U_p are shared while the rest are used by the PE's parent. If $R_l > 0$ and $R_r \leq 0$, which means that (i) the left child subtree has more idle PEs, and (ii) the right child subtree has more tasks to be consumed, the excess tasks from the right child subtree and the parent can be dispatched to the left child subtree. The case $R_l \leq 0$ and $R_r > 0$ is symmetric to the previous case. When both child subtrees have more tasks than idle PEs, which is equivalent to $R_l < 0$ and $R_r < 0$, the excess tasks from both child subtrees together with the tasks from the parent are dispatched to the PE's parent. When $R \leq 0$ at the root, there are still tasks to be consumed. When $R > 0$ at the root, there are more idle PEs than the tasks to be consumed. When *R* at the root equals the number of PEs in the system, meaning all PEs are idle, the termination has been reached. The SAV algorithm is classified as DBST.

Signaling Condition	Value Submitted
The processor does not want to delete from nor insert into the shared queue.	0
The processor inserts an element into the shared queue.	-1
The processor deletes an element from the shared queue.	0
The interface processor fails to delete an element from the shared queue.	+1
PE is idle.	+1

Table 3. SAV value returned by PEs.

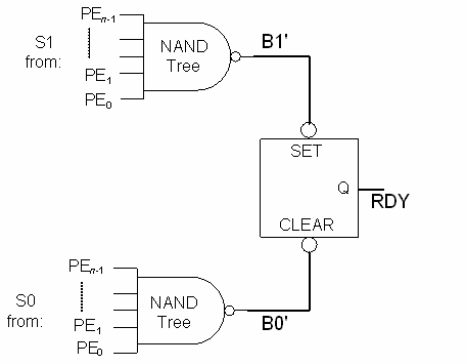

Figure 14. NAND tree in TTL_PAPERS.

$U_l \leftarrow U_r \leftarrow 0$;
$R \leftarrow R_l + R_r$;
case
$R_l > 0$ and $R_r > 0$:
$U \leftarrow min(U_p, R)$;
$U_l \leftarrow \left| \frac{R_l}{R} \times U \right|$;
$U_r \leftarrow \left| \frac{R_r}{R} \times U \right|$;
$R_l > 0$ and $R_r \leq 0$:
$U_l \leftarrow min(U_p - R, R_l)$;
$R_l \leq 0$ and $R_r > 0$:
$U_r \leftarrow min(U_p - R_l, R_r)$;
endcase
$R \leftarrow R - U_p$;
Send R to parent;
Send U_l to left child;
Send U_r to right child;

Figure 15. The SAV algorithm [17].

It is dynamic-binding because the joining PEs do not have to be known in advance. It is same-tasking because PEs can only be reactivated for the same barrier.

2) The Counting Algorithm

The *Counting Algorithm* [18] is a two-phase distributed termination detection algorithm. A copy of the algorithm runs on every PE. All participating PEs are organized as a spanning tree. Every PE keeps track of the created and processed tasks locally with the variables n_c and n_p respectively while it maintains the accumulated counts of the created and processed tasks of the subtree rooted at itself with the variables N_c and N_p respectively. In phase 1, each leaf PE sends the idle message with N_c and N_p initialized to n_c and n_p respectively after it turns idle. The idle message signifies that each PE in the subtree below has been idle at least once since the last idle message. This is contrary to the activity message in phase 2 which merely reports creation and processing activities. The other PEs update the local N_c and N_p by adding the N_c and N_p values sent

from their children with the idle message. After receiving idle messages from all its children, a PE sends an idle message with N_c and N_p, updated with n_c and n_p respectively, to its parent when it turns idle. When the root node has received idle messages from all children and turns idle, it compares N_c and N_p. If they are equal, it enters phase 2 since there is a high probability that the termination has been reached. Otherwise, it restarts phase 1. In phase 2, every PE sends up an activity message containing new values of N_c and N_p. These activity messages are assembled in the same way as in phase 1. When the root has received activity messages from all children nodes, it compares the old and new values of N_c and N_p. If they are equal, no new activities have occurred. The root declares termination of the barrier then. Otherwise, it restarts phase 1. The Counting Algorithm is classified as DBST. It is dynamic-binding because the joining PEs do not have to be known a-priori. It is same-tasking capable because PEs can only be reactivated to the same barrier. While the counting algorithm detects termination in wired distributed networks, new termination detection protocols have been proposed for wireless networks in which mobile nodes can be readily connected to and disconnected from the network at any time. In [19], it is assumed that the network consists of a first set of processes running on static hosts, a second set of processes running on mobile hosts, and a third set of processes running on base stations. Communication to and from a mobile process must be relayed through a base station. The transmission range covered by a base station is represented by a cell. The authors propose a hybrid approach in which they apply the LTD algorithm on each cell while the *Credit Algorithm*, proposed in [1] and described in Section IV.D.1 of this paper, is applied on the wired part of the network consisting of the first and third set of processes. The base stations work in between to bridge both algorithms.

C. Dynamic-Binding Different-Tasking (DBDT) Capable Techniques

1) The Wired-NOR Barrier

The Wired-NOR Barrier is a distributed and hardwired barrier architecture which supports both intracluster and intercluster synchronization [20]. An example supporting four barriers with two PEs is shown in Figure 16. In general, there are m barrier wires where each wire supports independently a single barrier. Physically, every barrier wire is connected to n PEs where n is the size of the system or cluster. Each PE i, $1 \le i \le n$, uses a *control vector* $X_i = (X_{i,1}, X_{i,2}, \ldots, X_{i,m})$ and a *monitor vector* $Y_i = (Y_{i,1}, Y_{i,2}, \ldots, Y_{i,m})$ for synchronization control. These vectors are mapped into the shared memory or distributed to special registers in each processor board making them program-accessible from each PE. Each barrier wire j, $1 \le j \le m$, is connected to n NPN bipolar transistors associated with n PEs separately. Alternatively, the barrier architecture can be conceived as a structure in which every PE contains m transistors tied to m barrier wires. In each PE i, the base of each transistor is connected to a control bit $X_{i,j}$ while the collector of the same transistor is monitored by a monitor bit $Y_{i,j}$. When a barrier comes into existence, the corresponding barrier wire is pulled up to the high voltage. Any PE sets its corresponding control bit $X_{i,m}$ when it enters the barrier. This makes the associated transistor closed and pulls down the voltage on the barrier wire. A PE resets its corresponding control bit when it finishes its job for the barrier. A barrier line will be pulled high again only when all transistors connected to the line are reset low. This reset action can be sensed by the monitor bit at which point the barrier is terminated. The Wired-NOR barrier architecture is classified as DBDT. It is dynamic-binding because the participating PEs do not have to be known in advance. It is different-tasking because PEs can only be reactivated to different tasks. A similar hardware approach for supporting barrier synchronization in parallel loops has been proposed by Beckmann and Polychonopoulus [21].

D. Dynamic-Binding Any-Tasking (DBAT) Capable Techniques

1) The Credit Algorithm

The Credit Algorithm is a global quiescence detection algorithm based on a very simple principle [1]. When the underlying computation begins, the controller, which can be assigned to either a dedicated or non-dedicated PE, distributes a credit of total value 1 to all processes. These processes either distribute part of their credit share to the new processes spawned by them or return the credit share to the controller when they finish or become *passive*. The controller declares termination when it regains all the credit. To ensure credit distribution, the algorithm complies with the following rules:

(i) When a process becomes passive, it transmits its credit share to the controller.

(ii) When an activating message with credit share C arrives to an *active* process, C is transmitted to the controller.

(iii) When an activation message with credit share C arrives to a passive process, C is transferred to the activated process.

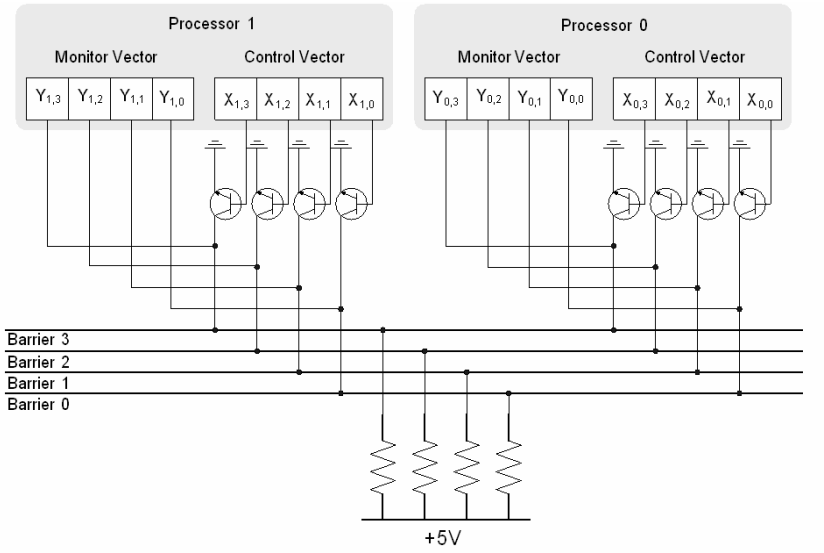

Figure 16. Wired NOR barrier.

(iv) When an active process with credit share C sends an activation message, the process keeps $\frac{1}{2}C$ and the message gets the other half.

Although not mentioned in the original paper, the Credit Algorithm can easily support multiple barriers by attaching a barrier ID to each credit share. Hence it is classified as DBAT. By the simple principle of credit distribution coupled with a clear lack of restrictions, it is obvious that the Credit Algorithm is DBAT. The Credit Algorithm can be readily incorporated into a negotiation protocol between multiagents and mediators. In a multiagent negotiation model, agents will make offers while mediators notify the agents of current pricing through message exchange between the agents and the mediators [22]. In this distributed asynchronous environment, the negotiation process is organized as successive iterations of strengthening commitments between the agents and the mediators. Deals are finalized when the negotiation process reaches a quiescent state. Since this state is a global property of the entire system, it can be detected by the Credit Algorithm.

2) The Tiered Algorithm

The basic approach of the Tiered Algorithm works by designating a node as the centralized *controller* to keep track of the global status [23]. Every joining PE reports the numbers of its consumed and produced tasks to the controller whenever it turns idle. The controller updates the records it keeps accordingly. After all PEs have become idle by reporting their tasks, the controller checks whether the global consumption count and production count match. If they do, it declares global termination. Otherwise, it waits for the next round of checks. However, the controller can sometimes make false detection based on the consumption and production counts. A scenario for a system with two PEs is shown below:

(i) PE_1 initiates local processing.
(ii) PE_2 is idle and reports to the controller: *produce* = 0, *consume* = 0, and *status* = *idle*.
(iii) PE_1 sends a process-create message to PE_2.
(iv) PE_2 receives the message from PE_1, sends a message to inform the controller of its active status, and begins processing which generates a new process-create message from PE_2 to PE_1.
(v) Before the controller processes the status update from PE_2, PE_1 receives the process-create message, completes its processing, and reports to the controller: *produce* = 1, *consume* = 1, and *status* = *idle*.

The controller can detect a false end-of-processing because it is aware of only one created process, one completed process, or is still considering both PEs to be idle. To prevent false detection, hierarchy information is employed instead of global information. The consumption and production counts are accounted for each level of the execution tree. The termination has been reached when the counts of consumption and production for each level match in the sense that $\sum_i n_{produced} = \sum_i n_{consumed}$ for every level i of the execution tree. The Tiered Algorithm can support multiple barriers by attaching a barrier ID to each message.

Mechanism	Classification	Multiple-Signaling Capability	Processor Reactivation Capability	Interconnection Network Limitation
Butterfly	SBIT	No	Yes	
U-cube Tree	SBIT	No	Yes	Wormhole-routed Hypercube
CV	SBST	Partial	Yes	
LTD	SBST	Yes	Yes	
CS Tree	SBDT	No	Yes	2D Mesh Network
F & A	SBDT	No	Yes	
AND Gate	DBIT	No	No	
TTL PAPERS	DBIT	No	No	
SAV	DBST	Yes	Yes	
Counting	DBST	Yes	Yes	
Wired-NOR	DBDT	No	Yes	
MBRH	DBDT	No	Yes	
Credit	DBAT	Yes	Yes	
Tiered	DBAT	Yes	Yes	

Table 4. Functionality summary.

Algorithm	Required Messages
Tiered	E
Credit	T
CV	$T + E + 2N - 1$
LTD	from $(E + N - 2)$ to $(T + N - 2)$

Table 5. Comparison of message complexity.

Based on the simple counting principle, the Tiered Algorithm has no restrictions at all. It can easily support dynamic process binding and multiple barriers. Consequently, it can be classified as DBAT.

V. SUMMARY OF SYNCHRONIZATION APPROACHES

A. Summary of Functionality

The functionality of the algorithms introduced in this paper and their mechanisms is summarized in Table 4. The multiple-signaling capability of the CV algorithm is considered as partial although it can reactivate PEs for the same barrier at the beginning. However, it cannot reactivated a PE when that PE becomes idle after sending the terminate message.

B. Analysis Based on Performance Metrics

The required message transmission can be analyzed using an event-driven methodology [24]. Here we analyze the two most-capable algorithms in the DBAT category, namely the CV and LTD Algorithms, for comparison purposes. The reason for choosing these two algorithms is that they both can be easily extended to the SBAT category by adding barrier identification. Let T, N, and E be the total task number, the number of joining PEs for each barrier, and the number of events experienced by all joining PEs respectively. The analysis on the number of required messages is summarized in Table 5. In processor-centered algorithms, an event is defined as a situation in which tasks are allocated to a specific PE, a situation in which all tasks are executed by a PE, or a situation in which a PE becomes idle. It is required that every joining PE reports its status to the controller after it becomes idle. It is obvious that for an efficient algorithm, the event number E ought to be smaller than the task number T. According to Table 5, the Tiered Algorithm performs best, the Credit and the CV Algorithms perform worst, while the LTD Algorithm ranges in between. Since the Credit Algorithm provides a conceptual design without detailed implementation, it can be readily implemented as a processor-centered algorithm. Under a processor-centered implementation, its message complexity can be reduced to E because each PE sums credit values locally and needs only to transmit local credit sub-totals whenever it becomes idle.

VI. CONCLUSION

In this paper, we describe the barrier synchronization problem and the requirements for designing barrier synchronization mechanisms. Next, we propose a novel taxonomy for barrier synchronization mechanisms that classifies them into eight categories, namely SBIT, SBST, SBDT, SBAT,

DBIT, DBST, DBDT, and DBAT. The capability hierarchy based on these categories is also presented. Any barrier synchronization algorithm can be easily identified by this taxonomy and its capability decided by the capability hierarchy. In addition, performance metrics, functionality, and design features for the barrier synchronization mechanisms are introduced in the paper whereby one or two examples are given in every category with the exception of the SBAT category of which no examples could be found in the published literature so far. Moreover, analytical performance evaluations are applied on newer barrier synchronization algorithms in order to shed some light on the improvement they bring to performance over older ones. While these algorithms have been around for some time, significant progress in parallel and distributed computer architectures has been made in terms of processor speed and memory access. New efficient primitive instructions such load-linked and store-conditionals have been introduced to support fast atomic operations in these new parallel architectures. As architectural innovation continues to move forward, numerous scalable applications have been made available to the research community for studying the performance of barrier synchronization and detection termination algorithms. As a result, performance studies such as the one presented in [25] are urgently needed to understand the tradeoffs between software algorithms, hardware support, and computation scale as they relate to the new parallel architectures.

REFERENCES

[1] F. Mattern, "Global quiescence detection based on credit distribution and discovery," *Information Processing Letters*, vol. 30, no. 4, pp. 195-200, Feb. 1989.

[2] E. D. Brooks III, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 14, pp. 295-307, 1986.

[3] N. S. Arenstorf and H. F. Jordan, "Comparing barrier algorithms," *Parallel Computing*, vol. 12, pp. 157-170, 1989.

[4] D. Hensgen, R. Kinkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1-17, 1988.

[5] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207-221, Nov. 1998.

[6] K. Hwang and F. A. Briggs, *Computer architecture and parallel processing*: McGraw-Hill, 1984.

[7] W. E. Cohen, D. W. Hyde, and R. K. Gaede, "An optical bus-based distributed dynamic barrier mechanism," *IEEE Transactions on Computers*, vol. 49, no. 12, pp. 1354-1365, Dec. 2000.

[8] H. Xu, P. K. McKinley, and L. M. Ni, "Efficient implementations of barrier synchronization in wormhole-routed hypercube multicomputers," *Journal of Parallel and Distributed Computing*, vol. 15, pp. 172-184, 1992.

[9] S. Chandrasekaran and S. Venkatesan, "A message-optimal algorithm for distributed termination detection," *Journal of Parallel and Distributed Computing*, vol. 8, no. 3, pp. 245-252, Mar. 1990.

[10] T.-H. Lai, Y.-C. Tseng, and X. Dong, "A more efficient message-optimal algorithm for distributed termination detection," *Sixth International Parallel Processing Symposium*, 1992, pp. 646-649.

[11] J.-S. Yang and C.-T. King, "Designing tree-based barrier synchronization on 2D meshes networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 6, pp. 526-533, Jun. 1998.

[12] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee, "Four-ary tree-based barrier synchronization for 2D meshes without nonmember involvement," *IEEE Transactions on Computers*, vol. 50, no. 8, pp. 811-823, Aug. 2001.

[13] Y. Sun, P. Y. S. Cheung, and X. Lin, "Barrier synchronization on wormhole-routed networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 583-597, Jun. 2001.

[14] K. Ghose and D.-C. Cheng, "Efficient synchronization schemes for large-scale shared-memory multiprocessors," *International Conference on Parallel Processing*, 1991, pp. 153-158.

[15] H. G. Dietz, R. Hoare, and T. Mattox, "A fine-grain parallel architecture based on barrier synchronization," *International Conference on Parallel Processing*, 1996, pp. 247-250.

[16] T. Muhammad, "Hardware barrier synchronization for a cluster of personal computers," Purdue University, May 1995.

[17] K. H. Cheng and Q. Wang, "A simultaneous access design for idle processor reactivation and the detection of the termination of a parallel activity," *Journal of Parallel and Distributed Computing*, vol. 17, pp. 370-373, 1993.

[18] A. B. Sinha and L. V. Kale, "A dynamic and adaptive quiescence detection algorithm," University of Illinois at Urbana-Champaign, Urbana-Champaign.

[19] Y.-C. Tseng and C.-C. Tan, "Termination detection protocols for mobile distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 558-566, Jun. 2001.

[20] S. Shang and K. Hwang, "Distributed hardwired barrier synchronization for scalable multiprocessor clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 591-605, Jun. 1995.

[21] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," University of Illinois at Urbana-Champaign, Urbana-Champaign 986, Nov. 1990.

[22] M. P. Wellman and W. E. Walsh, "Distributed quiescence detection in multiagent negotiation," *4th International Conference on Multiagent Systems*, 2000, pp. 317-324.

[23] Y. Tseng, R. F. DeMara, and P. Wilder, "Distributed-sum termination detection supporting multithreaded execution," *Parallel Computing*, vol. 29, no. 7, pp. 953-968, Jul. 2003.

[24] Y. Tseng and R. F. DeMara, "Event-based methodology for performance analysis of termination detection schemes," *International Conference of Parallel and Distributed Systems*, Taiwan, 2002

[25] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, "Evaluating synchronization on shared address space multiprocessors: methodology and performance," *ACM Joint International Conference on Measurements and Modeling of Computer Systems*, 1999, pp. 23-34.