Barrier Synchronization Techniques for Distributed Process Creation

R. DeMara and B. Motlagh
Dept. of Electrical and Computer Engr.
University of Central Florida
Orlando, FL 32816-2450
Tel: 407-823-5916
E-mail: rfd@engr.ucf.edu

C. Lin and S. Kuo
Dept. of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089
Tel: 213-740-4477
E-mail: changhwa@gringo.usc.edu

Abstract

Synchronization techniques are proposed for algorithms which spawn processes remotely on loosely coupled processors based on run-time characteristics. The performance of the proposed synchronization schemes are measured on the iPSC/2 and SNAP-1 multiprocessors and their implementation cost is discussed. Results show that processes created dynamically throughout a distributed system can be synchronized at comparable overhead and cost to that required for fixed-location process creation.

Keywords: Barrier Synchronization, Loosely Coupled System, Message-Passing, Symbolic Processing, Spawn-in-Transit, Hypercube Networks.

1 Introduction

While efficient barrier implementation is essential for both numeric and symbolic processing, previous research has focused primarily on synchronization mechanisims for numeric applications with predicable behavior where synchronization requirements are known a-priori. In this paper, we present synchronization methods suitable for concurrent programs whose synchronization requirements are determined at run-time, e.g. symbolic and recursive applications.

1.1 Dynamic Process Synchronization

Consider barrier synchronization for the block of code shown in Figure 1. The requirement is to execute two application-level processes P1 and P2 in parallel and then perform a barrier synchronization before executing P3. In the general case, each process may be composed of multiple subprocesses which can themselves be executed concurrently on different physical processors. During *distributed process creation*, the subprocesses to be spawned and their location of execution are not known ahead of time because they depend primarily on the values of input data and run-time variables.

For example, a process-flow graph for Figure 1 is shown in Figure 2. The application processes P1 and P2 correspond to the top-level processes $P_1^{1,1}$ and $P_2^{1,1}$, respectively. Several levels of process creation may occur at run-time as shown. Execution is initiated by a set of m parent processes $P_i^{1,1}$ where $1 \leq i \leq m$. Each parent process may selectively spawn n 2nd level subprocesses labelled $P_i^{2,j}$ where $1 \leq j \leq n$. Additionally, each subprocess can spawn its own children $P_i^{k,j}$ at the kth level, and so on. The value of n, m, and k depend on data within $P_i^{k,j}$, the current state of $P_i^{k,j}$, and messages received from other concurrently executing processes.

For example, in Figure 2, execution is initiated by the $m = 2$ parent processes $P_1^{1,1}$ and $P_2^{1,1}$. $P_1^{1,1}$ spawns $n = 2$ subprocesses and hence must remain active until they both terminate. Each level spawns additional processes while its parent process must wait at the barrier before executing application process P3. Under these conditions, performing the barrier synchronization is equivalent to detecting the termination

```
cobegin;
    P1;
    P2;
coend;
P3;
```

Figure 1: Parallel Code Fragment.

0-8186-5602-6/94 © 1994 IEEE

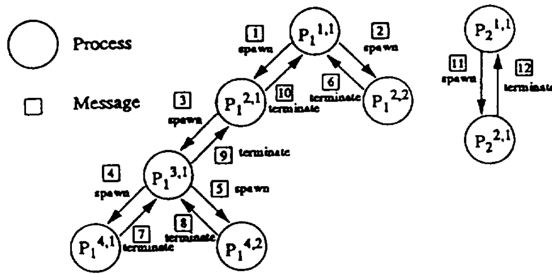

Figure 2: Dynamic Process Creation.

of all active processes throughout the system.

We assume a loosely-coupled multiprocessor with p physical processors. Given the total number of subprocesses $N = \sum_{i,k} n$, we also assume $N \gg p$, which is valid for parallel symbolic or recursive programs such as search tree traversal. At the system level, a parent process $P_i^{1,1}$ spawns a child process $P_i^{2,1}$ by sending a some type of *process create* message to the destination processor. Clearly, the number of processes created grows rapidly as the processing deepens.

1.1.1 Process-Centered Synchronization

Since enforcing the barrier is equivalent to verifying process termination, one can simply require each parent process $P_i^{1,1}$ to remain active until n *termination messages* are received, one from each of its children. This technique is illustrated in Figure 2. We refer to such an approach as *process-centered* since it reports terminations on a per-process basis. Unfortunately, this generates $m \times n_{ave}$ additional message traffic. This synchronization traffic can seriously degrade the communication bandwidth available for messages carrying live data associated with the actual computations.

Additionally, dynamic process creation may incur a *spawn-in-transit* effect. This refers to the situation when all processors are idle, yet there is a message in transit from one processor to another that will spawn a new subprocess upon arrival at its destination. While all processors appear to be idle, the barrier is not actually reached. These messages can be difficult to track, yet their proper accounting is vital for enforcing the barrier and thus ensuring correctness of program execution.

Spawn-in-transit messages may arise on any loosely-coupled architecture such as a message-passing multiprocessors like the iPSC hypercube, nCUBE, and others. Unfortunately, existing barrier synchronization algorithms do not adequately account for spawn-in-transit effects.

1.1.2 Processor-Centered Synchronization

An alternative to process-centered protocols are *processor-centered* methods. Processor-centered methods are especially suitable for dynamic process creation where $N \gg p$, as typically occurs in large concurrent programs which generate many spawned processes. Essentially, we relax the process-centered requirement that explicit termination messages are sent back to each parent process. Instead, process creation/termination counts are reported on a per-processor basis and a centralized total is maintained. This results in lower synchronization message traffic and overhead.

1.2 Previous Barrier Mechanisms

Three different approaches have been employed [1]:

1. **Symmetric Entry/Exit:** synchronization depends on each parent process keeping track of only the children it spawned so that there is perfect symmetry between process creation and termination. This technique is employed in Figure 2. While this approach does handle spawn-in-transit situations, it creates significant synchronization message traffic overhead.
2. **Spinlocks:** synchronization depends upon updating a shared variable which can be referenced only by atomic read/write accesses. The parent process first clears the spinlock and then spawns a subprocess. The parent then waits until the child process sets the argument to indicate completion. While this technique is ideal for master/slave execution, it can only support one subprocess per spinlock.
3. **Predetermined Paths:** the entry phase of a barrier may be propagated along a predetermined pattern such as a tree or linear topology. In a tree structure, each processor would take the entry signal and then pass it along to the next lower level on the tree. Alternatively, the signal phase could be implemented as a broadcasting of reversal of polarity of some state indicator.

Additionally, fast and efficient hardware-only techniques have been proposed which address some of these synchronization requiements. O'Keefe and Dietz [8][9] have proposed dedicated hardware consisting of a barrier queue, an associative memory, and an AND tree,which is formed by ANDing together the one-bit synchronization line from each processor. While powerful, their method is primarily suitable for monoprogrammed multiprocessors, and the associated control logic may become complex if the number of processors is large.

Hwang and Shang [6] have proposed an efficient wired-NOR hardware mechanism whereby each processor has multiple synchronization lines. At compile-time, the barrier patterns are analyzed and specific synchronization lines are assigned. A small, fixed number of concurrent barrier points, or *degrees of multiprogramming*, are supported by allocating synchronization lines to particular threads.

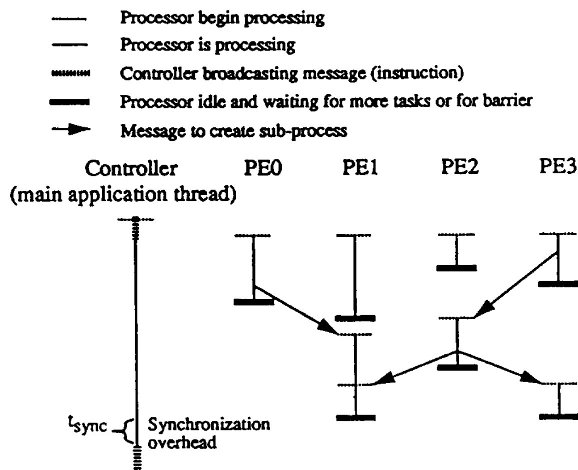

Figure 3: Definition of Synchronization Overhead.

Unfortunately, the above techniques are primarily suitable for programs where there is no spawn-in-transit behavior, i.e. the number and location of processes to be synchronized are known a-priori. In this paper, we propose methods for distributed-process-creating applications such as natural language understanding [11], speech recognition [7], search [5], and others involving recursion or remote procedure calls.

1.3 Evaluation Criteria

Performance criteria can be readily introduced via an example. Consider Figure 3 which depicts a space-time diagram for a four processor system. We would like to perform the barrier synchronization:

1. with low synchronization overhead time, t_{sync},
2. incurring only small degradation in communication bandwidth available to the executing processes,
3. supporting a large, variable number of active synchronization points, i.e. multiprogramming, and
4. remaining scalable in cost and performance as the number of physical processors grows large.

As shown in Figure 3, t_{sync} consists of the time from when the last subprocess terminates until the end of the barrier is recognized and indicated back to the main application thread. Essentially, this is the time elapsed from completion of last application instruction before the barrier until the first application instruction after the barrier.

Thus, it is useful to define t_{sync} as the *total overhead time* for synchronization, including all software and operating system overheads to more accurately reflect the true delay experienced by the application. Thus, our experimental results consist of actual measured synchronization time experienced by the application thread. This includes the time required to process any hardware signals up through the operating system to give the "barrier completed" signal back to the executing thread.

2 Tiered Protocol

A *tiered* synchronization mechanism is capable of handling spawn-in-transit scenarios on any multiprocessor system, either with or without specialized hardware support. Synchronization is performed by exchange of messages between PE's and a designated master node or *controller*. Since $N \gg p$, we reduce message traffic by reporting process creation/termination data on a processor-centered basis, regardless of the number of subprocesses created. We have executed our tiered protocol on the iPSC/2 hypercube multiprocessor for two benchmark applications and present the results below.

2.1 Termination Conditions

The controller node must determine if 1) all processing units are idle, and, 2) all communication channels are un-utilized. For the first condition, each processor has to send at least one message to the controller to indicate an idle state. For the second condition, it can be observed that the total number of messages created for spawning subprocesses must be equal to the sum of global subprocesses completed. So to detect the second condition, all processors may send the controller node their local subprocess *production counts* and *consumption counts* only when it has no jobs in its queue. For a p processor system, only $p \times i_{ave}$ messages are required for the synchronization task where i_{ave} is the average number of idle-state reportings to the controller corresponding to each empty queue condition. Experimental data reported in Section 4 indicates that $i_{ave} \leq 5$ for most applications.

However, even with the requirement of Condition 1, it is still possible to detect a false end-of-processing. Consider the following scenario for a two-processor system consisting of physical processors P_1 and P_2:

1. P_1 intiates local processing.
2. P_2 is idle and reports to controller: *produce* = 0, *consume* = 0, and *status* = *idle*.
3. P_1 sends a process-create message to P_2.
4. P_2 receives the message from P_1, sends a message to inform controller of its active status, and then begins processing which generates a new process-create message from P_2 to P_1.
5. Before the controller processes the status update from P_2, P_1 receives the process-create message, completes its processing and reports to the controller: *produce* = 1, *consume* = 1, and *status* = *idle*.

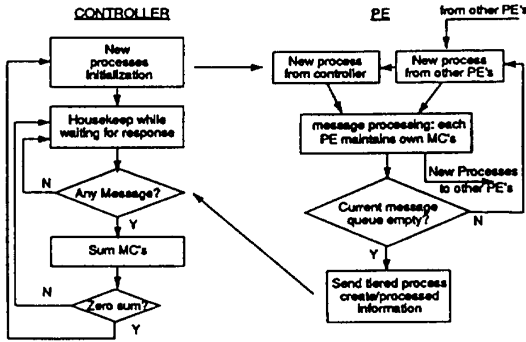

Figure 4: Tiered Synchronization Protocol

The controller node can detect a false end-of-processing because it is aware of only one created subprocess, one completed subprocess, while still considering both processors to be idle. To prevent this situation, once set, the idle status must be disabled before processing is resumed. In this case, P_2 must reconfirm with the controller before sending its spawn message from P_1. However, on a message-passing multiprocessor, the turn-around time could be long. The number of messages required becomes $p \times (i_{ave} + 2c)$ where the c is total number of confirmation requests. Normally, c is equal to $i_{ave} - 1$. Therefore, the message traffic is roughly tripled. Thus, the following tiered scheme is introduced to eliminate the confirmation requirement.

2.2 Produce/Consume Counting

Hierarchical information is employed to prevent false detection while reducing message overhead. The approach is to distinguish the different levels of process production and consumption. That is, for the k_{th} level, $\sum_i n_i^{spawned} = \sum_i n_i^{terminated}$ for each fixed k. For example, the global sum of first-level processes created must be equal to the number of first-level processes consumed. Therefore, the barrier detection ambiguity can be avoided since the process creation information is distinguished by depth of the subprocess level. Finally, to avoiding reporting separate spawn/terminate counts, each processor maintains a single local counter for each level. It has initial value of zero and is incremented upon each process creation and decremented after each process termination. The flowchart for this synchronization protocol is shown in Figure 4.

2.3 Execution Results

Two application benchmarks were executed on an 8 node iPSC/2 to measure synchronization overhead using the tiered protocol. The first benchmark called

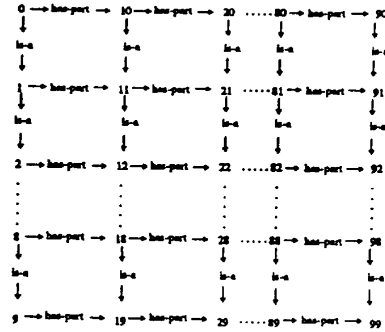

Figure 5: Benchmark #1: Inheritance Mesh.

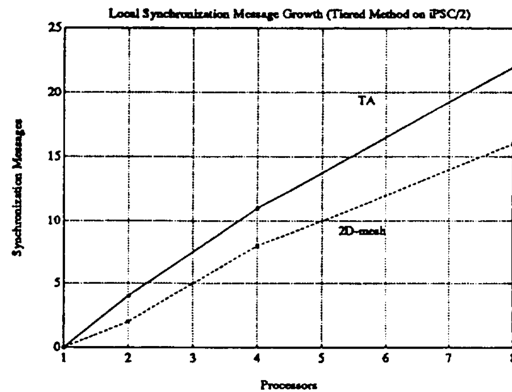

Figure 6: Local synchronization message growth (iPSC/2).

Inheritence – Mesh is depicted in Figure 5. It is a 2-dimensional grid where more specific concepts in inherit properties from more general concepts. A search algorithm is then applied to answer the query: `Does concept number 0 have property number 99?`

Processing originates at the physical processor which was allocated node 0 of the mesh. Processes are recursively spawned until node 99 is reached. The second benchmark called TA tries to answer the query: `Name all students who are single and live on campus.`

Figure 6 shows that the number of synchronization messages grows linearly as the number of processors is increased. Table 1 compares these results with the basic synchronization protocol described in Section 1. Synchronization message traffic has been significantly reduced. On average 84.7% and 93.6% less traffic was required. Total execution times and synchronization times are shown in Figures 7 and 8, respectively.

	Inheritence Mesh			TA		
# processors	Basic	Tiered	%reduction	Basic	Tiered	%reduction
2	17	2	88.2%	120	4	96.7%
4	49	8	83.7%	186	11	94.1%
8	90	16	82.2 %	222	22	90.1%

Table 1: Reduction in message traffic using tiered reporting (iPSC/2)

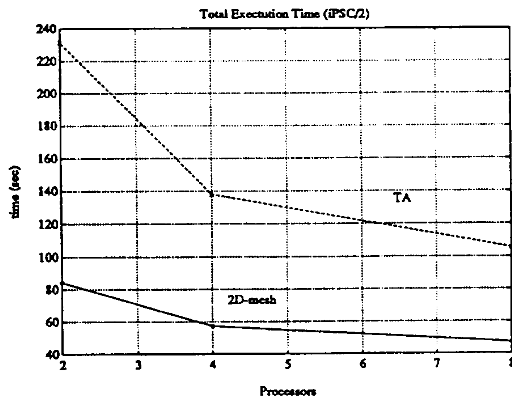

Figure 7: Total execution time (iPSC/2).

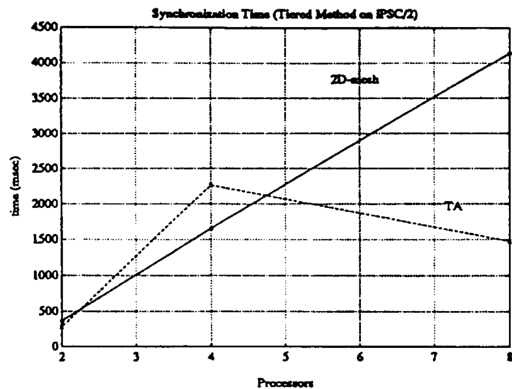

Figure 8: Synchronization time (Tiered method on iPSC/2).

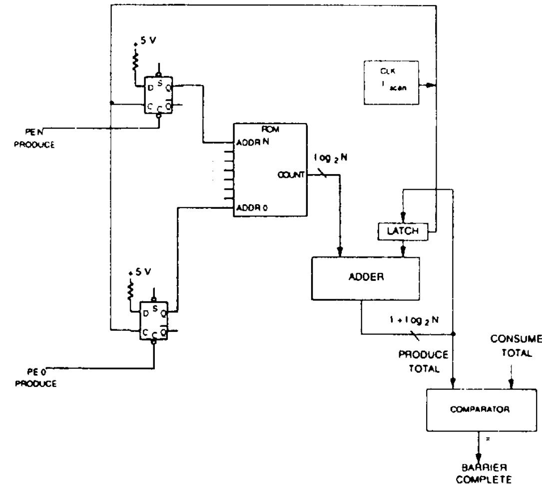

Figure 9: Central Sum Comparison Hardware.

3 Central Sum Comparison

Several hardware techniques based on variations of wired-logic mechanisms have been previously proposed for barrier synchronization [6], [8], [9]. While efficient, they do not accomodate spawn-in-transit effects. Thus, we propose a hardware design suitable for synchronizing algorithms which send messages to spawn processes on remote processors.

3.1 Hardware Design

The Central Sum Comparison hardware basically tabulates the global number of processes produced and consumed. These values are maintained in a centralized location and then continually compared to detect the completion of the barrier. Although, the discussion below describes only the subprocess *Produce Logic*, the design of the subprocess *Consume Logic* is identical.

As shown in Figure 9, one ROM chip, a quantity N of type D flip-flops, one adder, one latch, a clock, and a comparator are required. Alternatively, a simple PLA or combinatorial ASIC chip can be used in place of the ROM. Each processor is wired to the clock line of a D

flip-flop. This line is strobed once each time a process is produced. In the consume logic, a corresponding line is used to indicate termination of processes which is strobed every time a process is consumed by that processor.

The D flip-flops are used to prevent spurious reads of the same produce or consume signal. Thus, each flip-flop is clocked only once whenever the connected processor creates or terminates a process. The outputs of the all N flip-flops are connected to a ROM. These process create/terminate lines are viewed as addresses which refer to memory cells within the ROM. Each cell in the ROM contains a binary-encoded representation of the number of the active lines which are currently input to the ROM. This data is added to the value stored in the latch. The value in the latch represents the cumulative number of processes created up to that instant. Note that if no lines are active then a value of zero is added to contents of the latch so the cumulative sum remains unchanged.

Since processors are executing asynchronously, a separate clock is used to latch the results from the adder. Thus, an addition operation is perfomed at a predetermined rate denoted by f_{scan}. Let f_{strobe}^{max} be the strobe rate which corresponds to the maximum subprocess production rate. Then scanning frequency should be selected such that $f_{scan} >> f_{strobe}^{max}$. Note that each addition operation also clears the value in the D flip-flop to reset the system prior to the next cycle.

Finally, the cumulative number of processes produced and consumed are sent through a comparator. Since each parent processor will assert its produce line prior to spawning a child process, we can guarantee that the consume count always lags the produce count. Thus, instananeous equality between produce and consume counts indicates global process termination and thus completion of the barrier.

3.2 Synchronization Speed

The synchronization speed for the Central Sum Comparison method is determined by two factors. The first is the time required to generate a global synchronization signal. This is determined by the propagation delay of the synchronization logic and places an upper bound on f_{scan}. Assuming only moderately fast adders and comparators, synchronization could easily be detected within 100 nanoseconds, yielding a peak 10×10^6 synchronizations per second (10 MSPS). With currently available technology, this high level of performance is only realizable for approximately 32 processors due to the input-pin limitations of the ROM, PLA, or ASIC.

However, $32^2 = 1024$ processors can be easily accomodated using the hardware described above along with $\log_2 32$ or 5 tiers of adders organized into a tree structure. This would increase synchronization delay by an amount of $5 \times t_{add}$ or about 150 nanoseconds.

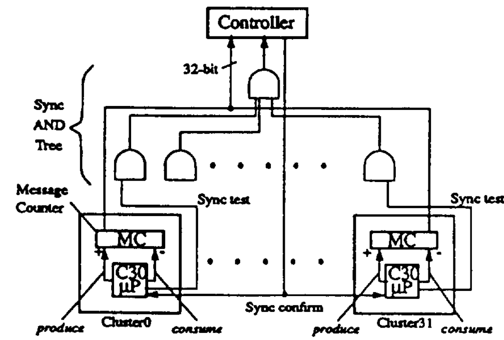

Figure 10: Synchronization hardware for SNAP-1.

Thus a 1024 processor machine can be synchronized in about 250 nanoseconds total, even if dynamic processes are created and spawn-in-transit messages exist. This is comparable to recently proposed approaches requiring 540 nanoseconds which do not accomodate spawn-in-transit scenarios [6]. Furthermore, potential circuit loading problems due to high fanouts are avoided.

In the next two sections, a hybrid synchronization algorithm is described. It shows that while it is slower than the Central Sum Comparison hardware, it incurs an acceptable performance degradation for some applications. Furthermore, the hardware requirement is reduced to a minimum while remaining scalable in the degree of multiprogramming.

4 Hybrid Synchronization

To obtain a balance among synchronization criteria, we have also designed and built a hybrid hardware/software synchronization mechanism for distributed process creation. We have implemented it within the Semantic Network Array Processor-1 (SNAP-1). SNAP-1 is a parallel machine for Natural Language Understanding and other Artificial Intelligence (AI) applications. It consists of 32 processing clusters, each with five functionally dedicated Digital Signal Processors (DSPs) [3].

The synchronization mechanism implemented in SNAP-1 employs the tiered protocol plus a minimum amount of specialized hardware as shown in Figure 10 to significantly reduce t_{sync}. The AND-tree provides a mechanism to interrupt the controller node. The dual-port memory which was already available for instruction buffering is used for transferring subprocess produce/consume counts. The flowchart for the protocol is shown in Figure 11.

Figure 12 shows the synchronization overhead for applications executed with the hybrid mechanism on

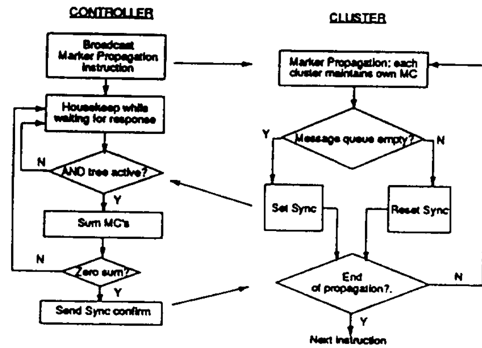

Figure 11: Synchronization Protocol for SNAP-1.

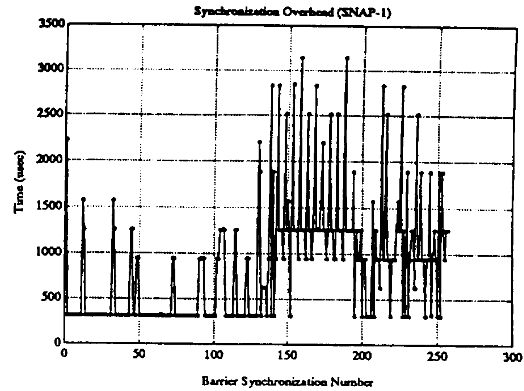

Figure 12: Synchronization Overhead (SNAP-1).

SNAP-1 for the Inheritance Mesh benchmark. A 16 cluster (80 processor) system was used with 22.1 MHz controller clock speed and 25 MHz array clock speed. The average values for synchronization overhead were 1250 and 856 μseconds, for Inheritance Mesh and TA benchmarks respectively. On average, 3.97 and 2.89 iterations were required for the Inheritence Mesh and TA benchmarks, respectively. Maximum observed values were 11 iterations while the median values were 4 and 3 iterations. The observed synchronization overhead time was still reasonably small, roughly 3.5%, compared to the overall application processing time.

5 Conclusion

We have developed three barrier synchronization methods suitable for dynamic process creation: a generally-applicable *Tiered Algorithm* which can be implemented solely in software on message-passing machines, a very high speed *Central Sum Comparator Method* using dedicated hardware, and a *Hybrid Scheme* which uses wired-logic to detect idle status and a software summation loop to handle spawn-in-transit messages. The Central Sum Comparison Method has the lowest synchronization overhead and is readily scalable in *log* space in terms of the number of processors, but not scalable in terms of the number of synchronization points. The message-based protocol is flexible in the number of synchronization points. By combining inexpensive dedicated hardware with processor-oriented message reporting through a hybrid approach, adequate synchronization time may be obtained for certain applications at very low cost. Furthermore, each of the techniques presented are capable of perfroming barrier synchronization efficiently even in the presence of spawn-in-transit messages.

References

[1] N. Arenstorf and H. Jordan, "Comparing Barrier Algorithms," *Parallel Computing* **12** (1989) pp. 157-170.

[2] E.D. Brooks III, "The Butterfly Barrier," *Int. Jour. Parallel Programming*, vol. 15 no. 4, pp. 295-307, 1986

[3] R. F. DeMara and D. I. Moldovan, "The SNAP-1 Parallel AI Prototype," in *Proceedings of Eighteenth Annual International Symposium on Computer Architecture*, Toronto Canada, 1991.

[4] D. Hensgen, R. Finkel, U. Manber, "Two Algorithms for Barrier Synchronization," Int. Jour. Parallel Programming, vol 17., no. 1, pp. 1-17, 1988

[5] L. Kanal and V. Kumar, Ed., *Search in Artificial Intelligence*, Springer-Verlag, 1988

[6] K. Hwang and S. Shang "Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessors," *Proceedings of the 1991International Conference on Parallel Processing*, 1991.

[7] H. Kitano, "DM-Dialog: An Experimental Speech-to-Speech Dialog Translation System," *Computer* **24**(6), pps. 36-50, June 1991.

[8] M. T. O'Keefe and H. G. Dietz "Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM)," *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[9] M. T. O'Keefe and H. G. Dietz "Hardware Barrier Synchronization: Static Barrier MIMD (SBM)," *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[10] M. R. Quillian, "Semantic Memory", PhD Dissertation, Carnegie Institute of Technology (Carnegie Mellon University), 1966.

[11] Y. Yu and R. F. Simmons "Truly Parallel Understanding of Text", *Proceedings Eighth National Conference on Artificial Intelligence*, 1990.

This document is an author-formatted work. The definitive version for citation appears as:

R. F. DeMara, B. S. Motlagh, E. Lin, and S. Kuo, "Barrier Synchronization Techniques for Distributed Process Creation," in *Proceedings of the Eighth International Symposium on Parallel Processing (IPPS'94)*, pp. 597 – 603, Cancun, Mexico, April 26 – 29, 1994. Inspec Accession Number: 4695877

Link:

http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=288243&isnumber=7174&punumber=958&k2dockey=288243@ieeecnfs&query=barrier+synchronization+techniques+for+distributed+process+creation&pos=0&arSt=597&ared=603&arAuthor=DeMara%2C+R.%3B+Mortlagh%2C+B.%3B+Lin%2C+C.%3B+Kuo%2C+S.%3B