Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Parallel Computing 29 (2003) 953–968

PARALLEL
COMPUTING

www.elsevier.com/locate/parco

Distributed-sum termination detection supporting multithreaded execution

Y. Tseng [a,1], R.F. DeMara [a,1], P.J. Wilder [b,*]

[a] *Department of Computer & Information Sciences, Florida A & M University, Tallahassee, FL 32307-5100, USA*

[b] *College of Engineering, Indiana Institute of Technology, 1600 East Washington Boulevard, Fort Wayne, IN 46803, USA*

Received 31 October 2001; received in revised form 16 December 2002; accepted 25 February 2003

Abstract

A fast, wire-efficient synchronization technique is developed that supports dynamic allocation of multiple threads on shared-memory, message-passing, and/or single-chip multiprocessors. The proposed distributed-sum bit-comparison (DSBC) method employs the execution-sequence invariant property such that the instantaneous process production equals the instantaneous process consumption only upon barrier completion. For a system of n processing elements (PEs), a single instance of the global logic unit, and n instances of the *local logic unit*, interconnected by $3n$ wires, are shown to provide direct support for any arbitrary number of barriers. The barrier detection time is shown to scale linearly in terms of the number of active barriers in the system. Comparisons to Wired-NOR hardware and Shared-Lock software approaches indicate reduced barrier detection time, decreased inter-PE wiring requirements, and increased functionality. Suitability of adaptation of the DSBC method to a skew-insensitive clockless design is also discussed.

Keywords: Multiprocessor barrier synchronization; Termination detection; Message-passing architectures; Shared-memory architectures; Quiescence detection

* Corresponding author. Tel.: +1-260-422-5561.
E-mail addresses: demara@mail.ucf.edu (R.F. DeMara), pwilder@indtech.edu (P.J. Wilder).
[1] Tel.: +1-407-823-5916; fax: +1-407-823-5835.

0167-8191/03/$ - see front matter
doi:10.1016/S0167-8191(03)00062-0

1. Introduction

Efficient barrier synchronization and quiescence detection techniques are fundamental to optimizing throughput in multiprocessor architectures. An ensemble of processing elements (PEs) is said to be *synchronized* [1,2], or to have reached a *quiescent state* [1], upon the completion of each concurrent-activity interval. The instances during execution when concurrent threads require synchronization are referred to as *barriers* [1–3]. These barriers must be detected prior to the resumption of processing to avoid violation of any precedence relationships between processes. Maintaining low overhead during barrier detection, while at the same time minimizing the hardware and software resources required for detection, is of primary importance in the development of an efficient barrier completion scheme.

1.1. Characteristics of barrier mechanisms

Characteristics that influence the selection of an appropriate barrier mechanism include the granularity of the threads in the application, the number of simultaneous barriers during execution, and the thread creation/allocation strategy used by the application. The *thread granularity* refers to the amount of processing that occurs during a concurrent interval. As threads become finer-grained, the relative impact of synchronization overhead on overall throughput becomes further magnified. Thus, frequently synchronized threads are less able to tolerate barrier detection latency, and are mostly likely to benefit from efficient hardware and software solutions to the barrier synchronization problem.

Single-threaded parallel applications require at most one barrier at any instant, while *multithreaded* parallel applications can take advantage of concurrent barriers to increase aggregate throughput and processor efficiency. For example, in a multi-user environment, each user's processes participate in distinct barriers. Since there are no data dependencies between processes from different users, multiple threads from one user can execute simultaneously along side of the threads of other users. However, the synchronization mechanism used must be capable of distinguishing between barrier signals used among different users' jobs. Likewise, single-user applications may also contain multiple threads participating in different active barriers that require distinct identification. As with the barrier method proposed in this paper, several recent research efforts have aimed at providing this important multiple barrier capability [14,15,17].

Other recent efforts have concentrated on efficient implementation of barriers when, at compile-time, knowledge of whether a particular PE participates in a barrier is not available [11,12]. In particular, applications that dynamically determine a PEs barrier participation, and/or generate new processes based on run-time conditions, are said to be capable of *adaptive process creation*. Multiprocessor applications requiring synchronization support for adaptive process creation include those with dynamically allocated remote procedure calls, recursive algorithms, interpreted program code, and others.

Additionally, barrier synchronization schemes may need to handle *launch-in-transit hazards*. This refers to situations when all PEs are idle, yet a message that is in transit between PEs creates a new sub-process upon arrival at the destination PE. Even if all processors are currently idle, the barrier cannot be attained when an additional process create operation is queued within the interconnection network. Thus, proper accounting of these messages is vital to barrier enforcement. Yet, this can be difficult to achieve with low overhead in a physically distributed computing environment, especially if processes are created adaptively at run-time. Launch-in-transit hazards can arise on distributed-memory architectures, LANs, or clusters of workstations. They are most difficult to deal with in environments where the synchronization technique cannot obtain an instantaneous snapshot of process status.

In summary, a functionally capable synchronization method should readily accommodate multiple threads, support adaptive process creation, and operate correctly in the presence of launch-in-transit hazards. Achieving this functionality in an efficient manner implies incurring low detection latency, while requiring minimal inter-PE message exchange and interconnection. The method proposed herein provides these capabilities at improved levels of performance in comparison to previous methods.

1.2. Previous work

The barrier synchronization issue has been studied from both software [1–9] and hardware perspectives [12–19]. A good overview of the barrier synchronization problem is presented by Arenstorf and Jordan [4], along with a representative range of solutions. In general, software approaches typically employ variations of Test-And-Set [10] or Fetch-And-Add [4] machine instructions. Hardware approaches include the *AND gate barrier* [13], a distributed TTL logic implementation called PAPERS [14], and a dedicated barrier synchronization Register Hardware [17]. More recently, Shang and Hwang presented a Wired-NOR logic design, with impressively low detection overhead [15,16]. In their paper, they also assess the degree to which hardware schemes can outperform software schemes.

However, even the Wired-NOR logic approach allows only one process per signaling wire to be dispatched to an individual processor, thus restricting multi-threaded applications. Additionally, Wired-NOR logic methods require that the process allocation to physical PEs be available at compile time. The distributed-sum bit-comparison (DSBC) logic developed in this paper is capable of overcoming the above limitations, while achieving termination detection latency comparable to that of Wired-NOR logic.

2. Distributed-sum bit-comparison technique

The DSBC logic configuration for n PEs is shown in Fig. 1. It includes one instance of *global logic* residing at either an independent node or any one local PE.

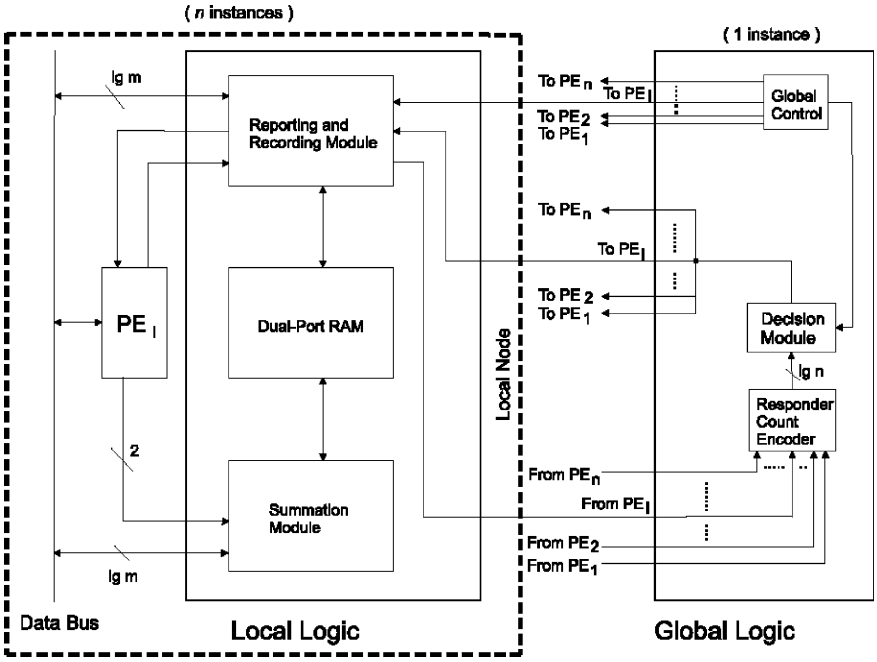

Fig. 1. Overview of DSBC layout.

The global logic unit communicates with n instances of *local logic*, whereby one instance of local logic resides at each PE.

The global logic unit consists of a *responder count encoder*, a *decision module*, and a *global control* signal source. Each local logic unit configuration is comprised of a *summation module*, a dual-port random-access memory (RAM), and a reporting and recording module (R&R). The DSBC technique employs local logic components to maintain a ledger of the number of processes produced or consumed by each thread at the local PE. These values are stored in a dual-port RAM for retrieval by the global logic unit. Thus, the approach is designated as a *distributed summation* technique, because each PE keeps a local ledger of process counts. It also employs *bit-comparison* methods, since the values in the ledgers are compared in a bit-serial fashion.

2.1. Operational concept

The global logic unit initiates Barrier detection. It requests the current process counts from the local logic units, and then sums them to evaluate whether the *barrier completion criterion* is satisfied. This criterion is satisfied whenever there exists an equal number of produced and consumed processes during any snapshot of system activity. It can be shown that this criterion is correctly computed even in the presence of adaptively created processes and launch-in-transit hazards [11].

Each PE notifies its own local logic unit whenever it produces or consumes a process. The local logic unit then makes adjustments to the local process count of the related barrier, which is stored in the dual-port RAM. To avoid the need to maintain distinct process production and consumption counts in the ledger, the process count for each thread is incremented before a process is produced, and decremented after a process is consumed. Thus, a dual-port RAM that is m words deep is capable of recording process counts to accommodate multithreading of degree m. The ledger RAM is dual-ported so that the global logic unit can simultaneously inspect the local process counts without impeding PE operation.

Once the global logic unit determines the completion status for a particular barrier, it returns the status to each local logic unit. Each local logic unit then responds by either storing the completed barrier number into a first-in first-out (FIFO) register, or advancing to inspect the next barrier. The completion information for the barrier is then available to the PE directly without incurring any unnecessary operating system overhead. Each of these system components is described in detail in Section 3.

2.2. Interaction between global and local logic units

As shown in Fig. 2, the global logic unit and local logic units operate in parallel. A technique is used to reduce both the required number of interconnections to each PE, and the resulting detection time. To reduce the wire count, a bit-serial scan is used. To reduce overhead that might otherwise be incurred by bit-serial communication, a useful mathematical property of progressive summation is employed as described below.

The first routine shown in Fig. 2 describes how each local logic unit acts to maintain the counts in its process ledger. The local logic unit increments the count for each barrier whenever a process is produced, and decrements it whenever a process is consumed. Initially, the process counts are set to zero. The procedure begins with the PE asserting the thread index on the data bus shown in Fig. 1. Then, the PE indicates process production or consumption by asserting the appropriate control line that serves as an input to the local logic unit. This minimizes the operating system overhead to simply toggling an I/O port, without incurring any delay-inducing competition for spin-locks [10] or other shared resources. This is a distinct performance advantage of using the local logic unit in the DSBC design.

The second routine shown in Fig. 2 describes the bit-scan process and the property of summation used by the global logic unit to skip over non-zero sums, quickly. Here the global logic unit considers each barrier in succession, and iteratively demands bits from the local process count ledgers for a specific barrier. Each PE responds with its own local process count for the designated barrier. The global logic unit keeps inspecting the global sums for consecutive barriers in a round-robin fashion, independent of other events. To calculate the global sums, it examines the counts bit-by-bit, starting from the least significant bit (LSB). If any bit of the sum is one, indicating a non-zero process count summation from all PEs is imminent, then the process count word for the next barrier is immediately fetched and checked. The reason for using this single-bit detection technique is as follows: a

```
Procedure DSBC();

  barrier_index=1; /* initialize to first barrier in the ledger */
  bit_index=1; /* initialize to first bit of the current barrier word */
  ledger[1..max_barrier_number]=0; /* initialize all ledger entries in all PEs */
  WORDSIZE=ceiling of log base 2 of maximum number of processes active at a PE for any
       barrier;

  parbegin  /* Local Logic and Global logic units operate in Parallel */

        begin  /* Local logic unit */
                repeat until FALSE {
                   if ( produced at PE by thread(i)) then ledger[i]++;
                   if (process consumed at PE by thread(i)) then ledger[i]--;
                } /* repeat indefinitely */
        end

        begin  /* Global logic unit */
                repeat until FALSE {
                    sum_value = sum of bits at location bit_index of ledger[barrier_index];
                    if (bit_index < WORDSIZE)
                                if (sum_value == 0) then bit_index++;
                                else
                                   bit_index=0;
                                   barrier_index++;
                    else
                                if (sum_value == 0) then
                                   bit_index=0;
                                   barrier_index++;
                                   write word_index in FIFO to indicate this barrier complete;
                                else
                                   bit_index=0;
                                   barrier_index++;
                } /* repeat indefinitely */
       end
 parend
```

Fig. 2. Local and global procedures in DSBC algorithm.

non-zero sum indicates that the barrier is not reached. It is likely that a non-zero bit will be encountered quickly, so that serial scanning is not necessarily inferior to the practice of comparing all bits simultaneously in parallel. Furthermore, during inspection, the barrier will more often be pending, rather than complete. Additionally, the local logic unit can potentially provide optimizations when all bits of a ledger entry are zero, implying a lack of meaningful barrier activity at that PE. Section 4 presents a detailed probabilistic analysis of the impact of these effects on performance.

3. Local logic components

Each of the local logic components shown in Fig. 1 cooperates to maintain the process ledger counts and to coordinate barrier status information as discussed below.

3.1. Summation module

The inputs from the local PE determine the action taken by the *summation module*. Fig. 3 shows the four possible activities, as well as the design of the summation module. Each activity is designated by a unique 2-bit code determined by two output lines from the PE. When the input code from the PE is 00, the summation module takes no action. When the input code is 01, indicating that the PE produced a process, the summation module reads the current process count from the dual-port RAM, adds one to the count, and stores the result in the latch. When the input is 10, meaning that the PE consumed a process, then the summation module reads the current process count from the dual-port RAM, deducts one from the count, and stores the result in the latch. The operation of deducting one can be executed by adding a value of negative one in two's complement form; for example adding 1111_2 to a 4-bit value. To simplify the control mechanism, an input of 01 or 10 can be assumed to be followed by a 11 input. The local logic unit generates this code, which enables the result stored in the latch to transfer into the dual-port RAM.

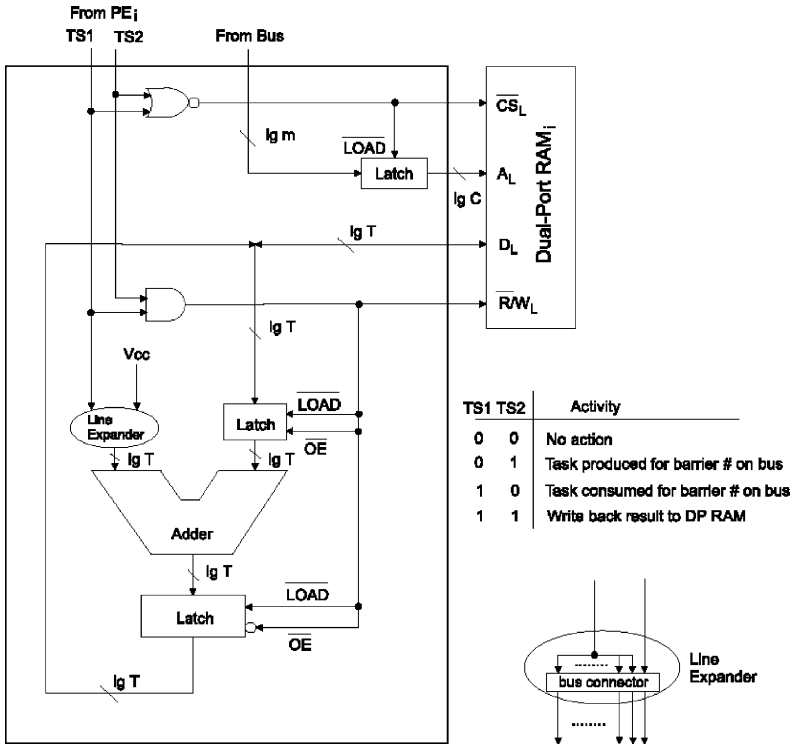

Fig. 3. Summation module.

3.2. Reporting and recording module algorithm

Fig. 4 shows the layout of the R&R module. The R&R module consists of two major components, a Parallel-In Serial-Out (PISO) register, and a FIFO register. The PISO register loads the process count in parallel, yet outputs the binary representation of the count serially, while the DSBC logic is inspects the count. The FIFO queue stores the completed barrier numbers, and is driven under the control of the global logic unit. The *global control signal* synchronizes the operation of the R&R module. Counter CNT1, shown in Fig. 4, controls the sequences used. A second counter, CNT2, maintains the next barrier number to be inspected, and provides the completed barrier number in the event that the current barrier is found to be complete. As the cycle begins, the R&R module loads the process count of the barrier (identified by CNT2) from the dual-port RAM into the PISO register. The PISO register outputs one bit at a time, starting with the LSB, to the global hardware. If the result from the decision module is zero, meaning more bits need checking to determine termination, then the next bit in the PISO register is fed to the global logic unit, and so on. If the result remains non-zero after the global logic unit checks all

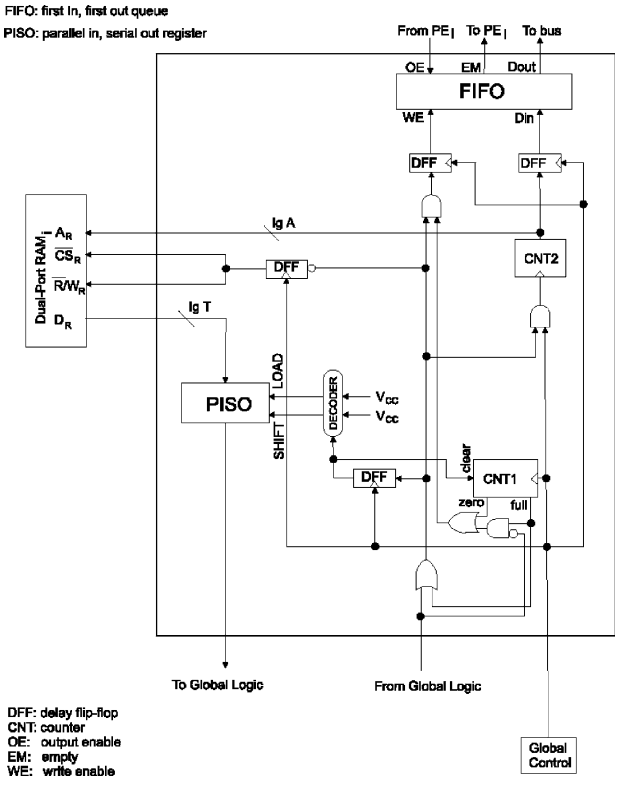

Fig. 4. Reporting and recording (R&R) module.

bits in the word, then the barrier is not complete, and the global control starts a new cycle with the next process count word for the next barrier. Alternatively, if, after bitwise summation, all of the resulting bits are zero, then the sum of all the local process counts is zero, implying that the barrier is finished. The control logic writes the current barrier ID into the FIFO queue, and begins a new cycle for the next barrier.

4. Global logic unit

4.1. Responder count encoder

The Responder Count Encoder sums the single-bit indicator lines from all PEs, and outputs the sum in a binary encoded format. As shown in Fig. 5, an adder-tree serves this purpose. $\mathrm{Log}_2 n$, the base-2 logarithm of the number n of supported PEs, bounds the number of levels in the tree. Additionally, a carry look-ahead design can provide a multiple-bit full adder capability without introducing significant gate delays. Other schemes involving carry-save representations are possible to mitigate the number of gate delays incurred for networks with a large number of PEs.

4.2. Decision module

Fig. 6 shows the *decision module*, which adds the sum from the responder count encoder to the carry output from the previous addition. It then extracts the LSB

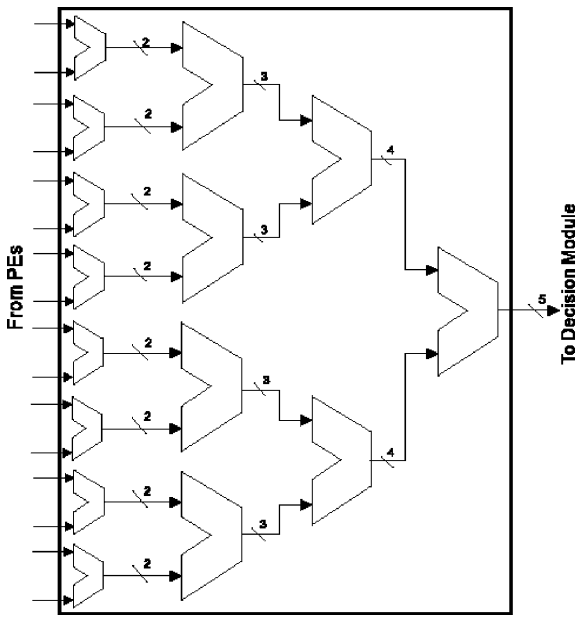

Fig. 5. Responder count encoder.

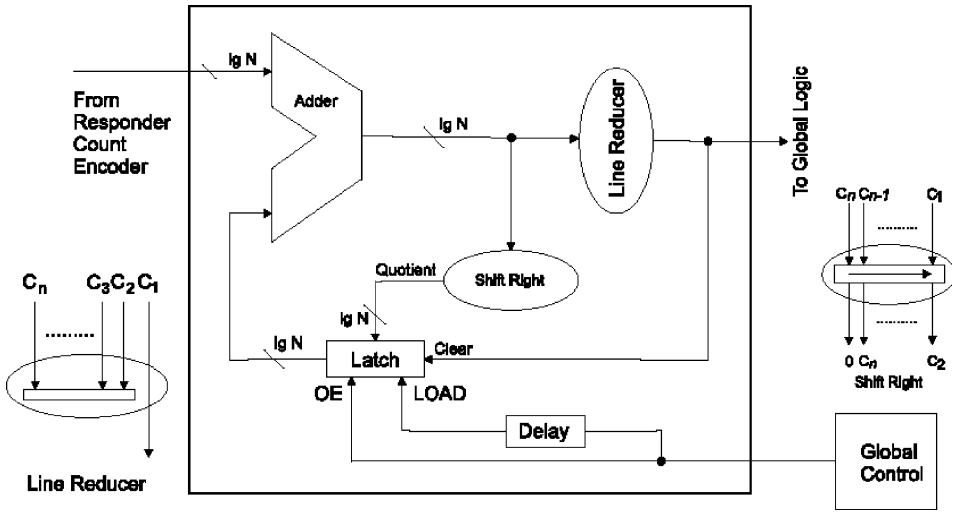

Fig. 6. Decision module.

to indicate the decision using the *line reducer*, as shown. The reason for adopting the LSB to indicate the result is that if the sum is an odd number then its LSB value is 1. The LSB clearly indicates immediate non-termination under this scenario. Alternatively, the result's LSB is zero for either even-numbered or zero sums. Under this circumstance, the hardware checks the next significant bit to decide whether the barrier is complete.

At the same time, the decision module directs the result from the adder to the *shift right function*, which can be implemented by just re-labeling each bit as the next less significant bit. This divide-by-two quotient is stored in the latch to be used as the carry for the next bit. However, the carry is cleared if the decision is one, since the carry only applies to the current barrier, and does not apply to the next barrier.

5. Performance analysis

This section first analyzes the time for termination detection with the DSBC logic, denoted T_{DSBC}. Next, it compares DSBC with a software-based Test-and-Set algorithm [10] and wired-NOR logic [15] in terms of performance and features.

5.1. Detection time

The procedures described previously for the detection of barrier completion can be decomposed as follows. First, the local process counts for the current barrier are written into the dual-port RAM after the barrier has been reached. Next, any completed barrier must wait its turn to be checked, requiring time T_{wait}. Once checking begins, the barrier can be determined to be complete, in serial-scan time denoted

by T_{word}. In particular, let this time denote the worst-case circumstance when all bits of the word, rather than just some of the bits in the ledger entry, are checked and stored in the latch in front of the FIFO queue. Finally, the barrier number is immediately written into the FIFO after the next cycle begins requiring time T_{FIFO}. Thus, this logic timing is given by Eq. (1).

$$T_{\text{DSBC}} = T_{\text{RAM}} + T_{\text{wait}} + T_{\text{word}} + T_{\text{FIFO}} = T_{\text{RAM}} + T_{\text{wait}} + q \cdot T_{\text{bt\_chk}} + T_{\text{FIFO}} \quad (1)$$

Here $q = t + w$, where t is selected so that 2^t bounds the maximum number of process counts supported by each PE, and w denotes the additional bits generated by the carry operations. Tseng and DeMara analyze the worst case time for the DSBC approach [20].

A nominal estimate the for DSBC parameters above can be found empirically using delay times from an implementation with discrete ICs, since a gate array or ASIC implementation would provide superior performance. First, the write time for the dual-port RAM, T_{RAM}, can be approximated at 30:10 ns to read the current count, 10 ns to write back the incremented count, and 10 ns to perform the addition. A nominal value for write time of the FIFO queue, T_{FIFO}, is 12 ns for 16-bit wide and 1,024 word deep FIFO. The wait time, T_{wait}, is variable and will be analyzed below. The time needed for one barrier-checkup cycle, T_{word}, ranges from 1 to q times the bit-checkup time, $T_{\text{bt\_chk}}$. For a completed barrier, all bits of the global process counts are zero; hence, each bit is checked to decide whether the barrier is reached. Thus, $T_{\text{word}} = q \cdot T_{\text{bt\_chk}}$, as indicated within Eq. (1). The bit-checkup time can be estimated by summing the cascading gate propagation delays along the critical path. However, the depth of the adder tree in the responder count encoder increases as the number of supported PEs grows. Therefore, a new notation $T^n_{\text{bt\_chk}}$, where n is the number of PEs in the system, is introduced to identify the different time delays depending on n. Even using standard discrete 7400-series components with fast-carry look-ahead, the bit-checkup cycle time for DSBC logic supporting $n = 256$ PEs is 190 ns or less.

As the DSBC logic relies upon an instantaneous snapshot of the system's state, the maximum propagation delays dictate a 190 ns cycle time to ensure data integrity. Because synchronization behavior of parallel applications varies widely, a probability-based estimate of the typical number of bit-checkup cycles in a barrier cycle provides a fair and equiprobable estimation. The calculation using an arithmetic-geometric series is given in Eq. (2).

$$\overline{T^n_{\text{word}}} = \left\{ \frac{1}{2} \cdot 1 + \frac{1}{2^2} \cdot 2 + \cdots + \frac{1}{2^{q-1}} \cdot (q-1) + \frac{1}{2^q} \cdot q \right\} \cdot T^n_{\text{bt\_chk}} \quad (2)$$

The first parameter in each term, except the final term, is the probability of getting a value of 1 from those bits remaining to be checked after encountering a string of i values of zero. The second parameter is the number of checkup cycles to be accounted for, which ranges from 1 to q. For example, in the second term the probability of encountering a bit with a value of one after already having encountered one value of zero is $(1/2) \cdot (1/2) = (1/2)^2$. This situation results in two bit-checkup cycles, because the first value of zero makes checking the next bit necessary. However,

the fact that the second value is one clears the need to check the next bit. The probability for the last term includes those values of both zero and one, because t bits are checked regardless of whether the bit value is zero or one. The rearrangement and summation of the series in Eq. (2) allows it to be expressed as Eq. (3):

$$\overline{T^n_{\text{word}}} = \left\{ 2 - \left(\frac{1}{2}\right)^{q-2} \cdot q + \left(\frac{1}{2}\right)^{q-1} \cdot (2q-1) \right\} \cdot T^n_{\text{bt\_chk}} \tag{3}$$

Assume that in the dual-port RAM m words are utilized, implying support for m barriers. The best case scenario is when the current barrier completes precisely when the previous barrier is finished being checked. The next barrier-checkup cycle detects the termination condition without waiting, hence $T_{\text{wait}} = 0$. The worst case scenario occurs when the barrier completes just after completion of checking. The termination of the current barrier cannot be detected immediately because the previous process count snapshot is being checked. Therefore, in the worse case, a delay of $(m-1)$ barrier-checkup cycles occurs before the barrier can be checked again.

Additionally, an equiprobable analysis provides a fair estimation of the typical wait time. The previous evaluation of the typical number of bit-checkup cycles in a barrier-checkup cycle serves the purpose, and is shown in Eq. (4):

$$\begin{aligned} \overline{T^n_{\text{wait}}} &= \left\{ \frac{1}{m} \cdot 0 + \frac{1}{m} \cdot 1 + \cdots + \frac{1}{m} \cdot (m-1) \right\} \cdot \overline{T^n_{\text{word}}} = \frac{(m-1)}{2} \cdot \overline{T^n_{\text{word}}} \\ &= (m-1) \left\{ 1 - \left(\frac{1}{2}\right)^{q-1} \cdot q + \left(\frac{1}{2}\right)^{q} \cdot (2q-1) \right\} \cdot T^n_{\text{bt\_chk}} \end{aligned} \tag{4}$$

The expected value for barrier detection time using DSBC logic can be obtained by substituting Eq. (4) into Eq. (1):

$$\begin{aligned} \overline{T^n_{\text{DSBC}}} &= T_{\text{RAM}} + \overline{T^n_{\text{wait}}} + T^n_{\text{word}} + T_{\text{FIFO}} \\ &= T_{\text{RAM}} + \left\{ (m-1) \left[1 - \left(\frac{1}{2}\right)^{q-1} \cdot q + \left(\frac{1}{2}\right)^{q} \cdot (2q-1) \right] + q \right\} \cdot T^n_{\text{bt\_chk}} \\ &\quad + T_{\text{FIFO}} \end{aligned} \tag{5}$$

Thus, the barrier detection time scales linearly with respect to m, and is independent of, or at most only weakly dependent on, growth proportional to $\log n$.

5.2. Comparisons of performance and features to previous techniques

Based on Eq. (5) and information both from Anderson [10] and Shang and Hwang [15], the detection times for four approaches are plotted in Fig. 7. Curves are plotted for DSBC logic supporting 16 barriers, DSBC logic supporting 32 barriers, wired-NOR logic replicated for an arbitrary number of barriers, and the Test-and-Set software. The three hardware-based schemes outperform the software-based scheme in termination detection, as the number of PEs increases. The ratio of benefit increases from 10-fold for 20 PEs to 1000-fold for 512 PEs. Both wired-NOR logic

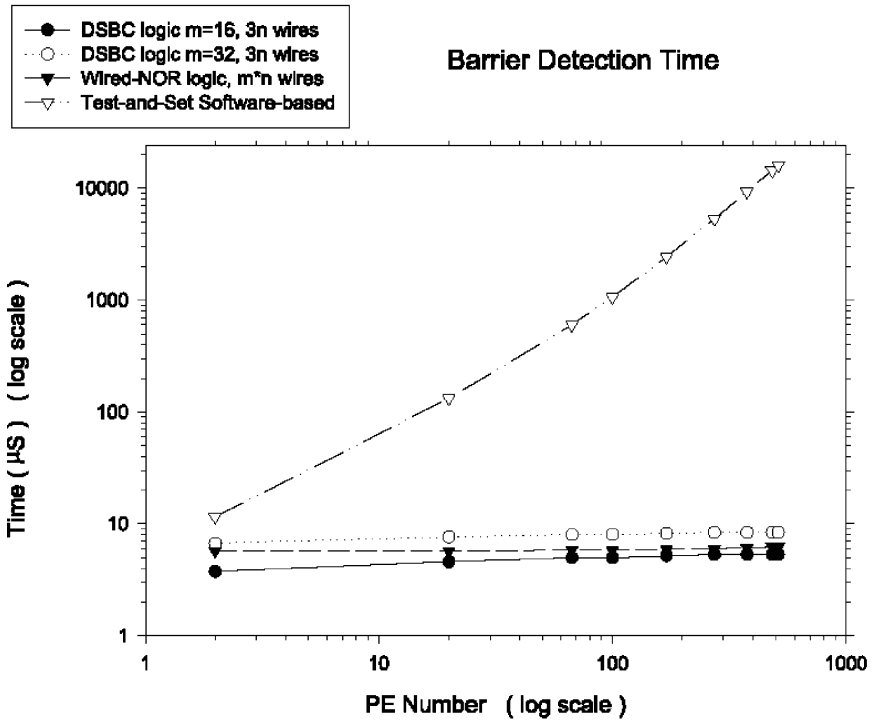

Fig. 7. Detection latency of representative barrier methods.

and DSBC logic have nearly constant detection times in the respective ranges of the number of PEs. Their detection times increase slightly, because new levels of propagation delay are added. In particular, to accommodate more PEs, additional levels of adders are required for the DSBC logic. In the case of wired-NOR logic, new repeater boards are required for larger numbers of PEs.

Theoretically, the detection time of the wired-NOR logic is independent of the number of barriers supported, m. However, m is restricted to 16 with current technology for wired-NOR logic [15]. DSBC faces no such restriction. Although the detection time of the DSBC logic increases as m increases, the version of the DSBC logic supporting 16 barriers takes less detection time than the wired-NOR logic, while the version of the DSBC logic supporting 32 barriers needs slightly more time than the wired-NOR logic. The latency for DSBC is conservative compared to wired-NOR logic because the DSBC estimates do not take into account speed enhancements available from semi-custom implementations, while wired-NOR logic assumes delays at transistor-level elements. Moreover, the DSBC logic requires only $3n$ lines between the local hardware and the global hardware (where n is the number of PEs in the system), while the wired-NOR logic requires $m \cdot n$ lines. Therefore, the DSBC logic still performs comparably, but at greatly reduced line complexity. Fig. 8 shows the comparison for the interconnect requirements.

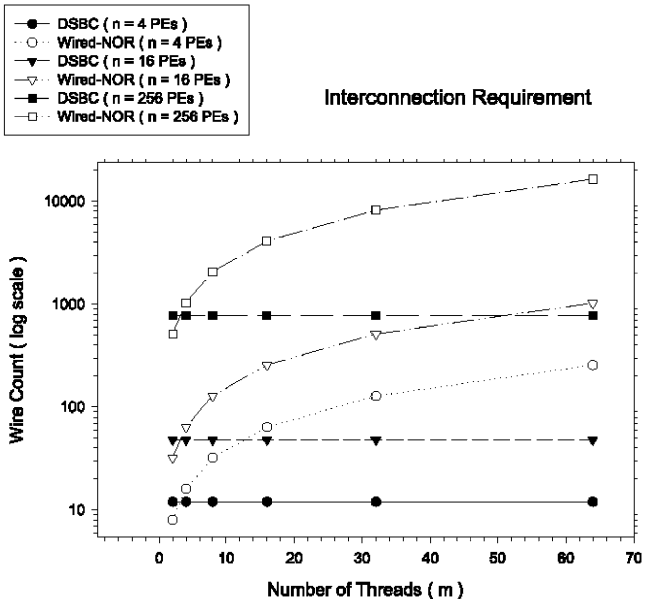

Fig. 8. Interconnection requirements of representative barrier methods.

There are additional features of the DSBC logic that are not revealed by the detection time comparison. The wired-NOR logic supports only one process, dispatched to each PE per signaling line, which is effective for computations that are statically scheduled and allocated at compile time. The DSBC logic can support execution of multiple processes, dispatched to each PE, which contribute to the same barrier dynamically at run time. This makes DSBC suitable for a wide range of applications. DSBC also provides adaptability, since the only interaction between the local PE and the DSBC logic is writing into the dual-port RAM and reading from the FIFO register of the local DSBC hardware. This fact makes applications on both message-passing architectures and shared-memory architectures easily adaptable to DSBC logic methods.

In some sense, the DSBC logic method blends aspects of traditional shared-memory and message-passing synchronization approaches. In the DSBC logic approach, the PE places an encoded message in the dual-port RAM to indicate production or consumption of a process. This action is similar to that of synchronization in message-passing architectures. However, the fact that the completed barrier number is stored in the local FIFO register, waiting to be inspected, acts somewhat like a shared-memory approach where the lock status is protected by atomic machine instructions. Moreover, both the operating system and the compiler can readily support the DSBC design, because multithreaded synchronization is implemented simply by accessing I/O ports or specific memory-mapped locations, thus avoiding the OS overhead associated with other approaches.

6. Conclusion

6.1. Summary

The DSBC technique supports dynamic allocation of multiple barriers, and multiple processes per barrier, while remaining scalable in terms of both detection delay and wiring complexity. DSBC can outperform some of the more efficient previous termination detection methods, while providing additional capability. The speedup of DSBC logic supporting 16 barriers over a Test-and-Set software-based scheme ranges from 29-fold for a 20-PE system to 3008-fold for a 512-PE system. With consideration of the number of barriers supported and the number of processes allocated to each PE, a very efficient customization of DSBC logic can be developed. Finally, DSBCs wiring requirement is less than that of wired-NOR methods for all configurations supporting more than 3 threads. The software interface to the DSBC logic consists of only writing to an I/O port and reading from a FIFO register. This relieves the compiler and programmer from the overhead of all synchronization activities except for accessing memory locations to obtain the identification number of all completed barriers.

6.2. Suitability in the presence of interconnection delay skew

Finally, the low wiring interconnection requirement of the DSBC strategy allows for a straightforward asynchronous adaptation. In physically distributed systems, or systems with large numbers of PEs, the interconnection delay between the PEs and global logic units can be significant. This can be remedied by converting only the 3 wires per PE to their dual-rail asynchronous encoding such as NULL convention logic (NCL). NCL uses two wires per bit to encode a TRUE, FALSE, and NULL (indeterminate) state for each bit, thus eliminating the need for clocked control signals. This approach is ideal for implementing barriers when a large number of PEs are involved, because signals are defined to have every DATA (either TRUE or FALSE) state flanked by a NULL state to eliminate both race conditions and the need for global timing information. This extension to clockless interconnect is unique among all previously proposed barrier detection approaches and is feasible due to the low wiring complexity of the DSBC design.

References

[1] F. Mattern, Global quiescence detection based on credit distribution and discovery, Information Processing Letters 30 (4) (1989) 195–200.
[2] E.D. Brooks III, The butterfly barrier, International Journal of Parallel Programming 15 (14) (1986) 295–307.
[3] D. Hensgen, R. Kinkel, U. Manber, Two algorithms for barrier synchronization, International Journal of Parallel Programming 17 (1) (1988) 1–17.
[4] N.S. Arenstorf, H.F. Jordan, Comparing barrier algorithms, Parallel Computing 12 (1989) 157–170.

[5] S. Chandrasekaran, S. Venkatesan, A message-optimal algorithm for distributed termination detection, Journal of Parallel and Distributed Computing 8 (3) (1990) 245–252.
[6] T.H. Lai, Y.C. Tseng, X. Dong, A more efficient message-optimal algorithm for distributed termination detection, Technical Research Report, OSU-CISRC-1/92-TR-2, Ohio State University, 1992. A brief version was published in Proceedings of Sixth International Parallel Processing Symposium, IEEE CS Press, March 1992, pp. 646–649.
[7] K.H. Cheng, 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 17 (1993) 370–373.
[8] H. Xu, P.K. McKinley, L.M. Ni, Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputers, Journal of Parallel and Distributed Computing 15 (1992) 172–184.
[9] J.-S. Yang, C.-T. King, Designing tree-based barrier synchronization on 2D mesh networks, IEEE Transactions on Parallel and Distributed Systems 9 (6) (1998) 526–533.
[10] T.E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessor, IEEE Transactions on Parallel and Distributed Systems 1 (1) (1990) 6–16.
[11] R. DeMara, B. Motlagh, C. Lin, S. Kuo, Barrier synchronization techniques for distributed process creation, in: 8th International Parallel Processing Symposium Proceedings April 1994, pp. 597–603.
[12] A.B. Sinha L.V. Kale, A dynamic and adaptive quiescence detection algorithm, Department of Computer Science, University of Illinois, Urbana, IL, 1991.
[13] K. Ghose, D.-C. Cheng, Efficient synchronization schemes for large-scale shared-memory multiprocessors, in: Proceedings of the 1991 International Conference on Parallel Processing, 1991, pp. 153–158.
[14] H.G. Dietz, R. Hoare, T. Mattox, A fine-grain parallel architecture based on barrier synchronization, in: Proceedings of the International Conference on Parallel Processing, August 1996, pp. 247–250.
[15] S. Shang, K. Hwang, Distributed hardwired barrier synchronization for scalable multiprocessor clusters, IEEE Transactions on Parallel and Distributed Systems 6 (6) (1995) 591–605.
[16] K. Hwang, S. Shang, Wired-NOR barrier synchronization for designing large shared-memory multiprocessors, in: Proceedings of the 1991 International Conference on Parallel Processing, 1991, pp. 171–175.
[17] C.J. Beckmann, C.D. Polychronopoulos, Fast barrier synchronization hardware, CSRD Report No. 986, University of Illinois, Urbana-Champaign, November 1990.
[18] H. Leung, H. Ting, An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems, IEEE Transactions on Parallel and Distributed Systems 8 (5) (1997).
[19] Y. Higashi, H. Chayamichi, M. Iwane, The range barrier synchronization mechanism on a bus-based multiprocessor, IPSJ Journal 38 (11) (2001).
[20] Y. Tseng, R.F. DeMara, Event-based methodology for performance analysis of termination detection schemes, in: 2002 International Conference on Parallel and Distributed Systems, Taiwan, ROC, December 17–20, 2002.