Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer

Nian-Feng Tzeng and **Angkul Kongmunvattana**

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Abstract

This paper introduces an efficient barrier synchronization algorithm based on the *binomial spanning tree* (BST) and proposes a data transfer reduction technique for distributed shared memory systems under release consistency. The introduced BST-based barrier algorithm parallelizes and distributes the workload amongs participating processors, alleviating network contention and yielding less retransmission. As a result, performance improves, and the degree of improvement increases quickly as the number of participants grows. Our barrier algorithm and data transfer reduction technique are incorporated in TreadMarks for evaluation using various benchmarks on a network of workstations and the IBM SP machine. Experimental results are gathered and demonstrated.

1 Introduction

Distributed memory systems usually exhibit poor programmability and portability, because all data partitioning and explicit communication must be done by the programmer. The distributed shared memory (DSM) has emerged to overcome this difficulty by providing a global, single address space on top of physically distributed memory systems. DSM combines the ease of shared memory programming paradigm with the scalability and constructability of distributed memory systems, such as the network of workstations (NOW) and distributed memory multiprocessors. Due to its potential advantages, DSM has been an active research area, with many prototype systems implemented and demonstrated [11, 13].

The address space of a DSM system is distributed across memories at interconnected processors. To reduce traffic over the network, a replication of certain data stored at a remote processor is usually created in the local cache of a processor. Multiple copies of data, while improving read performance, pose the need of coherence enforcement, which can be achieved through hardware, software, or a combination of the two [13]. Apparently, the software implementation is most attractive since it involves no added hardware, which tends to be machine-dependent and expensive. The data transfer due to coherent enforcement under a software implementation, however, has to be minimized. To this end, the release consistency (RC) model [5] is a preferred choice for a software DSM implementation due to its low coherence traffic. RC does not guarantee that shared memory is consistent all of the time, but rather making sure consistence only after synchronization operations. This naturally results in lower communication traffic due to coherence than a more restrictive memory consistency model, such as sequential consistency.

Barrier synchronization is a common and powerful primitive for synchronizing a large number of cooperating processors in a parallel system. It ensures *all* participating processors to reach a certain execution point before they may proceed beyond that point. Many papers on barrier synchronization have appeared in the literature [4, 6, 7, 10, 12], but none of them considered the particular need of the DSM system. Specifically, for software DSM systems under the release consistency model, barrier synchronization not only makes sure the arrival of all participants but also enforces memory consistency when they leave the barrier. As a result, all updates carried out by other processors have to be known or performed accordingly by a given processor at the barrier to guarantee coherence. Generally, different processors get different sets of updates. This makes it necessary for the DSM system under release consistency to "scatter" updates to all participants during barrier synchronization, as opposed to a simpler broadcast operation needed for shared-memory systems. We introduce and evaluate a binomial spanning tree-based algorithm for an efficient barrier implementation in this paper.

Memory initialization for TreadMarks is not always in a desirable manner, and excessive cold misses often result during the run time of an application. These cold misses cause a lot of unnecessary data transfer, which tends to hamper performance, in particular for those applications with short execution times. This is because a software DSM system, like TreadMarks, is very

ICS 97 Vienna Austria
Copyright 1997 ACM 0-89791-902-5/97/7..$3.50

sensitive to the amount of data exchanged in creating the shared memory abstraction. In this paper, we propose a simple technique which effectively avoids many cold misses, leading to improved performance.

2 Pertinent Background

In implementing the DSM system, various memory consistency models have been adopted. A less restrictive model tends to yield higher efficiency, because it normally imposes a shorter memory access latency and incurs lower communication overhead caused by memory coherence enforcement.

2.1 Release consistency

Release consistency (RC) [5] is one of the least restrictive memory consistency models, and it ensures a synchronized program to see a sequentially consistent execution through the use of two synchronization operations: *acquire* for a processor to get access to a shared variable, and *release* for a processor to relinquish an acquired variable, permitting another processor to acquire the variable. In a synchronized program, a processor is allowed to use a shared data only after acquiring it, and the acquired data may then be accessed and modified before being released (and subsequently acquired by another processor). Multiple updates on shared data are allowed locally by different processors, but all the updates have to be completed at any release following those updates.

An *eager* software implementation of release consistency can be found in Munin [3], where a processor delays propagating its updates of shared data until the execution of a release. Later on, a *lazy* release consistency (LRC) was considered [9], in an attempt to further postpone the propagation of updates until the acquire (to the shared data made by another processor). It has been demonstrated that LRC generally outperforms eager release consistency, with respect to the number of messages and the amount of data exchanged [9]. In this paper, we are interested in DSM systems under the release consistency model.

2.2 Barrier Synchronization

The barrier is a synchronization primitive for parallel computing. A participant of barrier synchronization typically involves three actions during each barrier: (1) posting its arrival at the barrier, (2) waiting for all other participating processors to reach the barrier, and (3) receiving a notification to proceed beyond the barrier. Early studies on barrier synchronization have focused on the use of specific synchronization hardware [7, 10], on the software implementation of synchronization algorithms [6, 12], or on the evaluation of barrier synchronization performance [4]. Those studies all deal with barrier synchronization in which action (3) is satisfied simply through a broadcast operation (or mechanism) to all participating processors. Under RC model, barrier synchronization provides coherent views of shared pages after the barrier. Every participant thus has to receive a message which contains update information needed for it to perform coherence enforcement when exiting from the barrier. Update information for every participant is generally distinct and is computed according to the update records gathered from all the arriving processors. As a result, the barrier under release consistency involves a "scatter" operation to forward different update information to different participants.

2.3 TreadMarks

TreadMarks [2] is a software DSM implementation under the LRC model, and our investigation is based on the TreadMarks framework. Each barrier in TreadMarks is associated with a processor, called the barrier manager, responsible for recording the arrival of processors and for notifying every involved processor as soon as the barrier is satisfied, using a message which contains the write-notice (i.e., update information) for the processor. Consider, for example, that processor 0 (denoted by P_0) is the manager of a given barrier. When a processor reaches the barrier, it sends its write-notice to P_0 and waits for a response from P_0. After all processors arrive at the barrier, P_0 gathers all the write-notices, and it then starts to create and sent a write-notice to each participant in sequence, notifying them to proceed beyond the barrier. In this case, P_0 may easily become the performance bottleneck, especially when the number of participants grows.

Interprocessor channels are established by means of the UDP/IP communication protocol, which is connectionless and fails to guarantee reliable message delivery. In the presence of contention, a message may be lost if an arriving message is handled by an interrupt service routine because the arrival of a message will not cause any interrupt, if an earlier arrived message is still processed by the interrupt service routine. Hence, a timeout mechanism is adopted to protect against unsuccessful message delivery. For example, after sending out its update record, a participant sets its clock and waits for a response. If the clock goes off before a response is received, the participant retransmits its update record and resets its clock. This process repeats until a response arrives. Retransmission as a result of this tends to decrease system performance considerably.

Memory Initialization

Memory management in TreadMarks is implemented using the OS segment violation (SIGSEGV) interrupt.

This system interrupt is handled by the service routine provided by TreadMarks, with the action in the routine determined by the cause of the interrupt and the current status of the memory page. The page status can be either valid or invalid, and can be either empty or non-empty. Also, each of the page sets its access mode to be read-only, read-write, or inaccessible, based on the status of the page. Therefore, OS generates an interrupt when invoking coherent activities is necessary.

In TreadMarks, every processor, except P_0, initializes all of its pages with the empty and invalid status; those pages are all set to the inaccessible mode. All pages in P_0, however, are set to the non-empty and valid status. Moreover, every byte of each mapped page in P_0 is initialized with zero, using a *memset* operation. As a result, all processors, except P_0, will experience cold misses when they fetch their local pages for the very first time. A cold miss is trapped by the OS and handled by an interrupt service routine. In this case, the service routine sends a request to P_0, asking for a copy of the page that causes an interrupt. In response, the whole page is transmitted over the network. A cold miss thus results in a large amount of data transferred, and the transmitted page often contains only zero values. A technique is presented later to reduce substantially data transfer due to cold misses.

3 Proposed Barrier Synchronization

As mentioned earlier, barrier synchronization under release consistency requires a "scatter" operation to deliver separate memory update data to all barrier participants, so that global memory coherence is enforced after leaving the barrier. A trivial barrier implementation, as found in TreadMarks, tends to create serious contention at the barrier manager and significant retransmission due to the message loss or the expiration of a waiting interval (for the response) in moderate and large systems. If the system size grows, an inefficient barrier implementation soon becomes the system performance bottleneck as a result of excessive barrier traffic overhead. This phenomenon indeed has been observed for the system size larger than 16 under certain benchmarks, as will be detailed in Section 5.

3.1 Approach

In contrast to designating a single processor for collecting update records, for calculating the update result for each participant and for scattering calculated update results to all participants in sequence, we propose an efficient barrier approach on the basis of a ***binomial spanning tree*** (BST) [8]. This BST-based barrier approach is found to be superior to other tree-based (such as the binary tree and other unbalanced trees) barrier

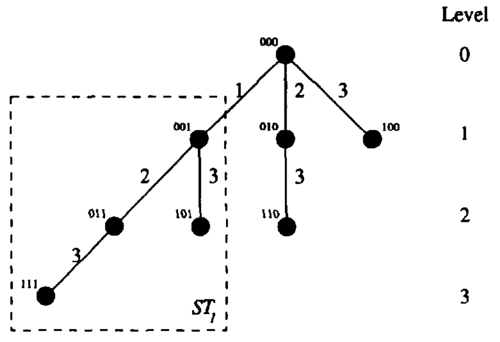

Figure 1: A binomial spanning tree (BST) of size 8.

schemes. A BST with four levels is illustrated in Figure 1, where the root is at level 0, its children are at level 1, and so on. This BST involves 8 nodes and the node degree ranges from 0 to 3, with one node having the degree of 3, one node having the degree of 2, and two (or four) nodes having the degree of 1 (or 0). In general, for a BST of 2^n nodes, one node is of degree n, one node is of degree n-1, and 2^{i-1} nodes are of degree n-i, $1 < i \leq n$. A subtree rooted at any node with degree j is referred to as ST_{n-j}, $0 \leq j \leq n$. In Figure 1, for example, the subtree rooted at node 001 (whose degree is 2) is ST_1 (for its subscript = 3-2), as enclosed in the dashed box. The subtree rooted at node 0 equals the whole tree, denoted by ST_0.

Nodes involved in a barrier are not necessarily contiguous, and the number of involved nodes is not always a power of 2. To facilitate our barrier implementation, the id's of all involved nodes are mapped to distinct logical id's ranging from 0 to N-1, where N is the total number of participants. A BST is constructed according to the logical id's of participants. Let a logical id (or id for short) be represented by a binary string $b_n b_{n-1} ... b_2 b_1$, where n is the smallest integer satisfying $N \leq 2^n$. Consider an ST_k ($k > 0$) and its parent node P in a constructed BST. If node P is addressed by $b_n ... b_{k+1} b_k b_{k-1} ... b_1$, then $b_k = 0$ and the root of ST_k is addressed by $b_n ... b_{k+1} 1 b_{k-1} ... b_1$. Additionally, the address of every node in ST_k, $a_n ... a_{k+1} a_k a_{k-1} ... a_1$, satisfies $a_k = 1$ and $a_i = b_i$ for all $1 \leq i < k$. In other words, the addresses of all the nodes in ST_k (including its root) have their rightmost k bits in common. This can be observed in Figure 1, where the addresses of all nodes in ST_1 have their rightmost bits in common. A formal definition of the BST is provided in [8]. In fact, a BST possesses above properties for any arbitrary N. The BST of size 10 is depicted in Figure 2 with solid lines, and it is easy to validate the properties for this BST. There are two ST_4's in the BST of size 10, as opposed to eight ST_4's in the BST of size 16 (shown by both solid and dotted lines in Figure 2).

Two of the three actions involved during each barrier benefit substantially from our proposed BST-based ap-

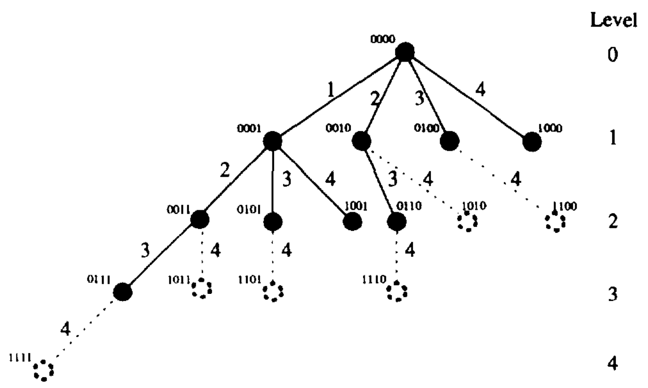

Figure 2: A BST of size 10 (shown by solid lines) and of size 16 (shown by both solid and dotted lines).

proach, i.e., posting arrivals at the barrier and preparing/sending release notification messages. In our barrier approach, a node, on reaching the barrier, does not send a message to notify its arrival at the barrier until its children all have reached the barrier. Every node sends a message (which also contains the memory update record) to its immediate parent to post its arrival at the barrier. This is similar in concept to the software combining technique described earlier in [14]. As a result, any node in a system with N participants will receive at most n such messages, where n is the smallest integer satisfying $N \leq 2^n$. Specifically, the root of any ST_j receives exactly $n - j$ such messages. If a shared medium (such as the Ethernet) is used for interconnecting nodes, there will be no more than $N/2$ participants competing for the medium initially with our approach, in contrast to N when a node is allowed to send its message off as soon as it reaches the barrier. This potential reduction in the degree of medium contention translates to a possible decrease in the number of messages retransmitted. In addition, since the node degree is limited to n, the probability of a message arriving during the interval when an earlier arrived message is still being processed (by an interrupt service routine) is significantly reduced, in comparison with the situation where the node degree is bounded by N. If a message arrives during such an interval, the message tends to be lost (since the interrupt is usually masked out during that interval), causing retransmission. As a result, our BST-based approach often lowers the chances of message retransmission considerably, in particular for a moderate or large N. This holds true for both shared and non-shared interconnecting media.

The third action of a barrier embraces preparing and sending distinct messages to all participants, as soon as the last processor reaches the barrier, to notify them to proceed beyond the barrier. This is referred to as the barrier release process, which is realized by a scatter operation under the release consistency model. The release process starts from the barrier manager (i.e., the BST root), which maintains all the memory update records. The BST-based approach minimizes the release process time, by allowing maximum possible parallelism in calculating the update results for all participants. Specifically, the BST root produces the cumulative update result for ST_1 and then sends the update result to the root of ST_1, denoted by $Root(ST_1)$, during step 1. At step 2, the BST root produces the cumulative update result for ST_2, while $Root(ST_1)$ calculates the update result for another ST_2 according to the received update result during step 1. These two update results are then sent respectively to the two $Root(ST_2)$'s. In general, 2^{j-1} nodes are involved in calculating separate update results for all the ST_j's simultaneously during step j. It takes n steps to complete the release of N ($\leq 2^n$) participants and that is optimal.

Our BST-based approach exhibits substantial performance improvement, when compared with the use of a single node for handling the barrier (like that found in TreadMarks). This implementation, however, may lead to a larger amount of data movement, because memory update records are accumulated at a parent node before forwarding upwards (and eventually reaching the BST root), and also cumulative update results are passed along to the root of a subtree during the release process. According to our measured results, the amount increase is negligible (less than 1%) for an Ethernet-based NOW of size 8, and it ranges from 7% to 24% for a distributed-memory parallel machine of size up to 32. However, the advantage resulting from retransmission reduction and parallelized computation/communication associated with our barrier approach well offsets the impact due to this increase, as will be demonstrated in Section 5.

3.2 Algorithm

Our algorithm starts with assigning distinct logical id's, $0, 1, ..., N-1$, to participating processors, with 0 always assigned to the root node, i.e., the barrier manager. Each processor then determines the id's of its parent and its children (if any), according to its assigned id, say id_{my}, for the construction of the BST. Specifically, the parent of id_{my} is obtained by resetting the leftmost bit "1" in the binary representation of id_{my}. On the other hand, the child(ren) of id_{my} is (are) computed by adding 2^i to id_{my}, where i satisfies $log_2(id_{my}) < i < \lceil log_2(N) \rceil$. If there are multiple i's satisfying the above inequality and the added result (i.e., $id_{my} + 2^i$) is smaller than N, node id_{my} has multiple children. Each node determines its parent and child(ren) during the initialization phase in the algorithm given below.

Upon arrival at the barrier, a processor waits for the check-in message(s) from its child(ren) to arrive before sending its update record to its parent. All update

Program	Problem Size	Sync. Type
3D-FFT	64x64x64	Barrier
Barnes	5k Particles	Lock, Barrier
SOR	2000x1000	Barrier
Jacobi	1200x1200	Barrier

Table 1: Benchmark Programs.

records from its children are combined together with its own update record into a single cumulative update record, which is then sent to its parent. This is called the *check-in phase* in the following algorithm. After checking in, a processor waits for a release message from its parent. The release phase is initiated by the root, immediately after it has received check-in messages from all its children. A release message is produced and sent to one of its children at a time in sequence. On receiving a release message from its parent, a node starts to generate release messages for its children one by one, as stated in the release phase of the algorithm below. It takes at most $\lceil log_2(N) \rceil$ steps to complete the barrier release process, an optimum result.

Algorithm: BST-based barrier algorithm

Initialization Phase

```
Assign distinct id numbers to all $N$ participating processors.
If (processor id == 0)
  Determine its children
Else
  Determine its parent and its children (if any)
EndIf
```

Check – in Phase

```
If it has any child
  Wait until all of its children to arrive at the barrier
  Accumulate update records from its children, then combine
      them with its own update record
  If (processor id != 0)
    Send the cumulative update record to its parent processor
  EndIf
Else
  Generate and send the update record to its parent processor
EndIf
```

Release Phase

```
If (processor id == 0)
  Generate and send release messages with distinct update
      information to its children in sequence
Else
  Wait for the release message from its parent
  Incorporate received update information in its data pages
  If it has any child
    Generate and send release messages with distinct update
        information to its children in sequence
  EndIf
EndIf
```

4 Performance Evaluation on NOW

We evaluate the performance of our proposed barrier synchronization by incorporating it in TreadMarks version 0.9.7, with the remaining TreadMarks codes kept unchanged. The problem sizes and the type of synchronization of benchmarks used are listed in Table 1.

4.1 Results and Discussion

The experimental results of the proposed barrier approach on a network of eight SUN workstations model SPARCstation2 running SunOS version 4.1.4 is presented in Table 2, Figure 3 and Figure 4. All the results are based on 30 independent runs, with an approximate 95% confidence interval for each provided value equal to the value ± 2%. Table 2 lists the total amount of data movement and the total number of messages passed over the network, which together reflect traffic due to data exchange and memory coherence enforcement during execution. The numbers of retransmitted messages given in the table indicate the degree of network contention and, consequently, the amount of messages lost during barrier operations. Our proposed barrier approach, when incorporated in TreadMarks, leads to noticeable reduction in the number of retransmitted check-in messages for 3D-FFT and Barnes benchmarks, as can be observed in those Ours(1) rows, because the total numbers of messages for the two benchmarks are large originally. Our proposed barrier approach alleviates the degree of network contention, thereby leading to a reduced number of retransmission.

The average barrier times (i.e., the average completion time per barrier) for the four benchmarks are depicted in Figure 3. The proposed barrier approach reduces the average barrier time in 3D-FFT by 21%, but achieves only 7%, 4.5%, and 8% reduction in Barnes, SOR, and Jacobi benchmarks, respectively (according to the Ours(1) bars). The percentage of barrier time reduction signifies the potential run time savings, as a result of our barrier approach. Because of a small sized NOW used, the number of retransmitted messages in the original TreadMarks is rather low (see Table 2). The proposed barrier approach can reduce only a few retransmissions for 3D-FFT and Barnes, but the total numbers of messages remain roughly the same as those in the original TreadMarks. For the other two applications, since the degrees of contention over the network are low (due to fewer total messages, as listed in Table 2) originally in TreadMarks, the proposed barrier approach does not lower the total numbers of messages, resulting in smaller amounts of barrier time reduction. While the amount of data movement is expected to be higher when our barrier approach is incorporated as mentioned earlier, the increased amount is always negligible (i.e., < 1%) for any benchmark studied, as far as

Program	Algorithms	Total Data Movement (KBytes)	Total Messages	Rexmited Check-In Messages	Rexmited Release Messages	Total Cold Misses
3D-FFT	TreadMarks	63398	33188	20	0	4263
	Ours(1)	63511	33177	14	0	4263
	Ours(2)	47297	25112	14	0	0
Barnes	TreadMarks	24583	49628	52	0	1071
	Ours(1)	24717	49608	42	0	1071
	Ours(2)	21918	43073	42	0	428
SOR	TreadMarks	7267	4309	7	0	1780
	Ours(1)	7300	4309	7	0	1780
	Ours(2)	338	4309	7	0	0
Jacobi	TreadMarks	5110	3311	0	0	1253
	Ours(1)	5149	3311	0	0	1253
	Ours(2)	180	2965	0	0	0

Ours(1): proposed barrier approach incorporated
Ours(2): proposed barrier approach and data transfer reduction technique incorporated

Table 2: Experimental results on NOW of size 8.

a system size of 8 is concerned. The degree of run time reduction due to the proposed barrier approach is not significant for every benchmark on NOW of size 8, as can be observed in Figure 4.

4.2 Data Transfer Reduction Technique

As mentioned earlier, the cold misses invoked by all processors but P_0 result in excessive data transfer over the network, jeopardizing performance. Next, we introduces a technique for overcoming this problem.

4.2.1 Technique

To eliminate cold misses, we propose to reset every byte of the mapped pages in each processor to zero during the initialization process (rather than leaving it empty and invalid). This initialization can be done by every processor individually using the *memset* operation and it does not increase any initialization time, because it can be done in parallel and at the same time when P_0 is carrying out its initialization. Every processor thus initializes all of its local pages with the non-empty status (rather than the empty status). In order to maintain memory coherence, only those pages in P_0 obtain the valid status while the pages in any other processor are still invalid. As a result, every processor, except P_0, experiences initially a miss that costs only the transfer of updated data (represented in the form of "diff"). The size of updated data is often much smaller than that of pages. A considerable amount of data transfer may thus be saved as a result of our proposed technique.

This implementation requires modifications in two program modules: page and heap initialization of TreadMarks. While the changes are very limited, noticeable performance improvement results, in particular when an application largely shares the memory pages but rarely shares the same set of data or variables. For such an application, the barrier typically serves as key

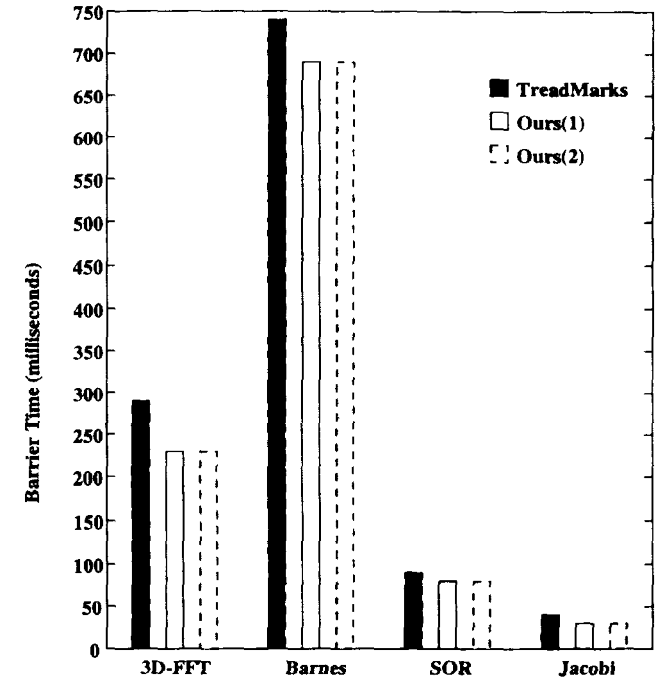

Figure 3: Average barrier time on NOW of size 8.

synchronization and significant reduction in data transfer is common. Moreover, the degree of improvement is boosted if each processor uses its local pages extensively during the run time. We have gathered the execution results of four benchmark codes executed under both original TreadMarks and modified TreadMarks on an Ethernet-based NOW system. Collected results are illustrated and discussed next.

4.2.2 Results and Discussion

The proposed barrier approach and our modification to memory initialization together lead to noticeable performance improvement when incorporated in TreadMarks, represented as Ours(2) in Table 2 and Figures 3 – 4. According to the experimental results listed in the table, our memory initialization technique reduces the total amount of data transfer by as much as 96% for SOR and Jacobi benchmarks, but achieves only 25% reduction for 3D-FFT and 11% for Barnes-Hut codes. The percentage of data transfer reduction indicates the level of decreased traffic possibly during run time, as a result of our memory initialization.

3D-FFT has the largest amount of data transfer due to cold misses, and our modification eliminates all the cold misses (see Table 2). The percentage of data transfer reduction, however, is not as high as SOR and Jacobi because some parts in most of the memory pages for 3D-FFT are shared by processors. Our modification thus leads to nearly the same number of diffs transmitted as the cold misses for 3D-FFT, but the diff size is much smaller than the page size.

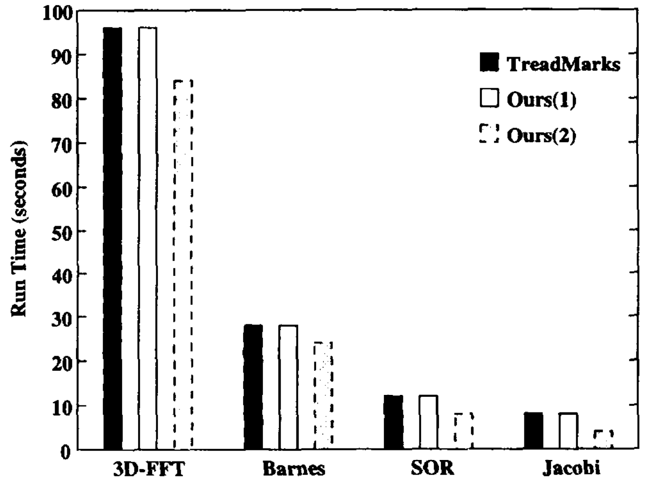

Figure 4: Run time on NOW of size 8.

Barnes has the fewest cold misses (i.e., only 1071 from Table 2), and therefore the percentage of data transfer reduction is not high. Moreover, the cold misses that happen after the storage reclamation routine is performed, is unavoidable. This routine cleans up all the write-notice records, twins, and diffs. Naturally, many cold misses occur thereafter.

SOR and Jacobi benchmarks exhibit high reduction ratios because they have relatively small amounts of data transmitted and many cold misses originally, as can be seen in Table 2. These cold misses occurred when processors use a scratch array to store the new values computed during each iteration, as mentioned earlier. The scratch array is allocated on the pages that are never used by any processor, including P_0. Therefore, the cold misses in the original TreadMarks cause P_0 to transmit the pages that contain only the zero values, but our modification eliminates such transmission altogether.

This reduction in data transfer translates to a decreased run time of the benchmarks, as illustrated in Figure 4. Specifically, 3D-FFT enjoys run time reduction by 11%, Barnes by 10%, SOR by 38%, and Jacobi by 37%. As expected, the degree of reduction corresponds to the amount of data transfer reduction.

5 Performance Evaluation on IBM SP

In order to evaluate the behavior of our proposed barrier approach in a larger system, we used the IBM SP machine [1] at Argonne National Laboratory as a hardware platform for study. The same set of four benchmarks is executed on this SP machine with TreadMarks employed. When the proposed barrier approach is incorporated, all the codes, except the barrier implementation, are kept unchanged, as before. The data transfer reduction technique presented earlier is not included in the study of this SP machine. The number of participants for 3D-FFT, SOR, and Jacobi is 32, but that for Barnes is only 18. Due to an excessive number of messages present in Barnes, the original TreadMarks never terminated successfully when the system size went beyond 18, whereas modified TreadMarks with our barrier incorporated still terminated successfully for a system size of 32. For comparison, however, we provide the results of Barnes on the SP machine of size 18 only for both TreadMarks and modified TreadMarks.

Program	Algorithms	Total Data Movement (KBytes)	Total Messages	Rexmitted Check-In Messages	Rexmitted Release Messages
3D-FFT	TreadMarks	99293	74090	6990	340
	Ours(1)	105947	62949	1803	23
Barnes	TreadMarks	51479	163965	813	9
	Ours(1)	55048	163512	530	8
SOR	TreadMarks	13661	9052	586	447
	Ours(1)	15887	7387	12	0
Jacobi	TreadMarks	8927	7601	349	188
	Ours(1)	11076	6742	15	0

Ours(1): proposed barrier approach incorporated

Table 3: Experimental results on IBM SP machine.

The total numbers of retransmitted messages get drastically reduced as a result of our proposed barrier approach, which lowers contention and quickens the barrier release process considerably, as listed in Table 3. From Figure 5, 3D-FFT is found to achieve 72% reduction in the average barrier time, SOR and Jacobi enjoy more than 80% reduction, and Barnes has about 32% reduction (which could have been much larger, had it been executed on a system of size 32). In original TreadMarks, all the participants register their arrivals at the barrier directly through the barrier manager by sending messages immediately when they are ready, causing severe contention and message losses even for a system size of only 32 (or 18 in the case of Barnes). The proposed barrier approach effectively alleviates this problem, yielding a big decrease in the number of retransmitted check-in messages (see Table 3). Similarly, the release process benefits substantially from the proposed barrier approach.

Under this system size, the communication time appears to dominate the total execution time, and the improvement in communication overhead due to the proposed barrier approach translates to a significant decrease in the run time, as demonstrated in Figure 6. Specifically, 3D-FFT gives rise to 41% run time reduction, SOR and Jacobi to 58% and 63%, respectively, but Barnes exhibits only 17% reduction since it is on a smaller system. While the proposed barrier approach always results in fewer messages transmitted for any benchmark, it tends to increase the total amount of data movement, as mentioned earlier. From Table 3, the increased amount in data movement due to our barrier approach ranges from 7% to 24%.

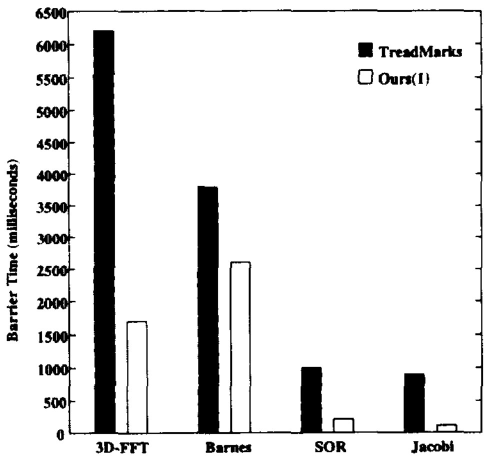

Figure 5: Average barrier time on IBM SP machine.

6 Conclusions

We have proposed an approach to barrier synchronization in distributed shared memory (DSM) systems. The experimental results demonstrate that our barrier approach effectively improves the performance of software DSMs under release consistency, especially when the system size grows. This is because more participants result in swiftly increased overhead due to barrier synchronization, with respect to both the number of retransmitted messages and the computation load associated with producing separate update information for participants during the barrier release process. The proposed barrier approach not only parallelizes the barrier process but also alleviates network contention, lowering the possibility of message retransmission. As a result, our barrier implementation makes it possible to construct a larger sized software DSM system, by removing a performance bottleneck likely to arise as the size increases. The proposed technique for data transfer reduction can eliminate many cold misses during program execution on TreadMarks, considerably shortening the run times of benchmarks examined.

Acknowledgements

This material is based upon work supported in part by the NSF under Grants MIP-9201308 and CCR-9300075. The authors gratefully acknowledge the use of the Argonne High-Performance Computing Research Facility, which is funded principally by the U.S. Department of Energy, Mathematical, Information and Computational Sciences Division (ER-31).

References

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

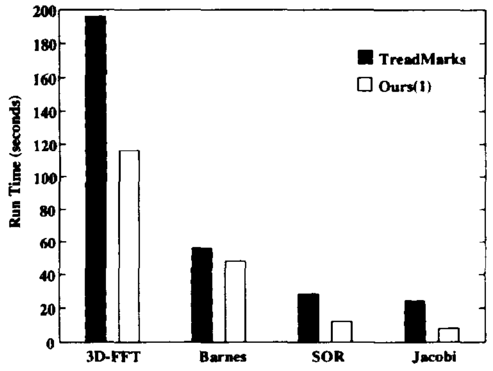

Figure 6: Run time on IBM SP machine.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[4] S. Y. Cheung and V. S. Sunderam. Performance of Barrier Synchronization Methods in a Multiaccess Network. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):890–895, August 1995.

[5] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[6] D. Hensgen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *Int'l Journal on Parallel Programming*, 17(1):1–17, February 1988.

[7] D. Johnson, D. Lilja, and J. Riedl. A Circulating Active Barrier Synchronization Mechanism. In *Proc. of the 1995 Int'l Conf. on Parallel Processing (ICPP'95)*, volume I, pages 202–209, August 1995.

[8] S. L. Johnsson and C.-T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans. on Computers*, C-38(9):1249–1268, September 1989.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[10] M. T. O'Keefe and H. G. Dietz. Hardware Barrier Synchronization: Static Barrier MIMD (SBM). In *Proc. of the 1990 Int'l Conf. on Parallel Processing (ICPP'90)*, volume I, pages 35–42, August 1990.

[11] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology*, 4:63–79, Summer 1996.

[12] M. Scott and J. Mellor-Crummey. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.

[13] N.-F. Tzeng and P.-C. Yew. Special Issue on Distributed Shared Memory Systems. *Journal of Parallel and Distributed Computing*, 29(2), September 1995.

[14] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. on Computers*, C-36(4):388–395, April 1987.