RICE UNIVERSITY

The Effect of Contention on the Scalability of Page-Based Software Shared Memory Systems

by

Eyal de Lara

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Houston, Texas

January, 1999

The Effect of Contention on the Scalability of Page-Based Software Shared Memory Systems

Eyal de Lara

Abstract

We demonstrate the profound effects of contention on the performance of page-based software distributed shared memory systems, as such systems are scaled to a larger number of nodes.

Programs whose performance scales well experience only minor increases in memory latency, do not suffer from contention, and show a balanced communication load. In contrast, programs that scaled poorly suffered from large memory latency increases due to contention and communication imbalance.

We use two existing protocols, Princeton's home-based protocol and the TreadMarks protocol, and a third novel protocol, Adaptive Striping. For most of our programs, all three protocols were equally affected by latency increases and achieved similar performance. Where they differ significantly, the communication load imbalance, which is caused by the read accesses to pages that have multiple readers following one or more writers, is the largest factor accounting for the difference.

Acknowledgments

I would like to express my thanks to the members of my committee, Dr. Willy Zwaenepoel, Dr. Alan L. Cox , and Dr. Sarita V. Adve, for their guidance and support in writing this thesis. I am convinced that without their assistance, this thesis would not have been possible.

I want to thank my advisor, Willy Zwaenepoel, for giving me a broader view of the work I was doing. His opportune advise steered me in the right direction and reminded me, in several occasions, not to loose sight of the forest while making my way through the trees.

I want to thank Alan L. Cox for his endless patience and support. I can not quantify how helpful those late night discussions were to the completion of this work.

I want to thank Charlie Hu for porting HLRC and helping me run and understand several of the benchmarks used in this study.

I like to thank my wife for her support and encouragement. It was certainly not easy to leave her side all those nights to come back to work to finish this thesis.

Last but not least, I would like to express my appreciation to my parents and brothers for always encouraging me to pursue my dreams.

Contents

Tables

Illustrations

Chapter 1

Introduction

Beginning with Li and Hudak [14] in 1985, designers of software distributed shared memory (DSM) protocols have focused much of their efforts on reducing the number of messages to maintain memory consistency. Indeed, protocol comparisons on small networks with eight to sixteen nodes have consistently shown a strong correlation between the execution time of a program and the number of messages exchanged by the protocol. This thesis shows that for many programs, on larger networks, the *distribution* of the messages is no less important than the number of messages. In other words, *which* node sends or receives a message and *when* it sends or receives that message have dramatic effects on the program's execution time.

Ideally, messages should be distributed uniformly in time and across nodes. Thus, avoiding contention for a node that consumes its processing time, delaying other computation, or contention in the network that delays message delivery. In some sense, a good distribution does not improve performance. It simply enables the nodes and the network to perform as one would expect under a simple performance model, in which, latencies for similar messages do not vary. A bad distribution creates load imbalance at the nodes and in the network that results in delays of computation and communication, adversely affecting execution time.

This thesis explores experimentally the relationship between the distribution of messages and performance on a network of thirty-two nodes for three page-based, multiple-writer protocols implementing Lazy Release Consistency (LRC) [11]. We use two existing protocols, Princeton's home-based protocol [20] and the TreadMarks protocol [12], and a third novel protocol, Adaptive Striping. Adaptive Striping is a simple extension of the TreadMarks protocol. Its goal is to eliminate load imbalance (within the protocol) at the nodes automatically.

We evaluate these protocols using a suite of five programs. They are Barnes-Hut from the SPLASH suite [18]; ILINK, which is a computational genetics code [9]; 3D FFT from the NAS benchmarks suit [3]; Gaussian Elimination and Red-Black SOR, which are small kernels. The communication in Red-Black SOR is balanced. The remaining programs exhibit common sharing patterns that lead to communication imbalances. We perform the evaluation on a switched, full-duplex 100Mbps Ethernet network of thirty-two 300 MHz Pentium II-based uniprocessors running FreeBSD 2.2.6.

This work has four specific conclusions. First, we show that for every program except SOR the time spent updating memory increases dramatically as the we extend the size of the cluster. In one case, it increases by 380%. This dramatic increase is not due to a growth in the number of messages, but the increasing time that each request for data takes. The result is poor scalability for these programs.

Second, we show that communication imbalance is the main factor responsible for this contention, especially with data strucutres with multiple readers.

Third, we found the performance of the home-based and TreadMarks protocols to be similar for all but two programs, where communication imbalance is the largest factor accounting for the difference. The imbalance is caused by the read accesses to pages that have multiple readers following one or more writers. For these programs, we found that the Adaptive Striping protocol achieved performance equal to, or better than, the home-based protocol without requiring the manual assignment of homes.

Fourth, the results for Adaptive Striping show that the distribution of the messages is no less important than the number of messages. For example, for Barnes-Hut, Adaptive Striping and the home-based protocol transfer roughly equal amounts of data and have identical message distributions. Although Adaptive Striping sends 12 times more messages than the home-based protocol, they achieve the same speedup.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of Lazy Release Consistency and the TreadMarks and Princeton multiple-writer protocols. Chapter 3 introduces contention, protocol load imbalances and describes the Adaptive Striping protocol. Chapter 4 presents the results of the evaluation of TreadMarks, the home-based protocol, and Adaptive Striping. It also includes details on the experimental platform that we used and the programs that we ran on it. Chapter 5 discusses related work. Chapter 6 summarizes our conclusions.

Chapter 2

Background

2.1 Lazy Release Consistency

Lazy release consistency (LRC) [11] is an algorithm that implements the release consistency (RC) [8] memory model. RC is a relaxed memory model in which ordinary accesses are distinguished from synchronization accesses. There are two types of synchronization access: acquire and release. Roughly speaking, acquire and release accesses are used to implement the corresponding operations on a lock. In general, a synchronization mechanism must perform an acquire access before entering a critical section and a release access before exiting. Essentially, the benefit of RC is that it allows the effects of ordinary memory accesses to be delayed until a subsequent release access by the same processor is performed.

The LRC algorithm [11] further delays the propagation of modifications to a processor until that processor executes an acquire. Specifically, LRC insures that the memory seen by the processor after an acquire is consistent with the happened-before-1 partial order [1], which is the union of the total processor order of the memory accesses on each individual processor and the partial order of release-acquire access pairs. Vector timestamps are used to represent the partial order [11]. When a processor executes an acquire, it sends its current vector timestamp in the acquire request message. In the reply message, granting the lock, the releaser includes a set of write notices. These write notices detail which shared data were modified prior to the acquire according to the partial order. These shared data modifications must be reflected in the memory as seen by the acquirer. In this paper, we consider invalidate protocols, in which the arrival of a write notice causes the corresponding page to be invalidated. On a subsequent access miss to an invalid page, the page is updated by requesting and applying all modifications described by the write notices for that page.

2.2 Multiple-Writer Protocols

Multiple-writer protocols were developed to address the false sharing problem. With a multiple-writer protocol, two or more processors can simultaneously modify their copy of a (shared) page. In accordance with RC's requirements, their modifications are merged at the next synchronization operation. By delaying the merger of the modifications, the ill effects of false sharing are reduced because the page will not ping-pong between the writers.

The first implementation of a multiple-writer protocol appeared in the Munin system [5]. Munin, however, used a form of *eager* RC that is generally inferior to LRC [10]. Since then two multiple-writer protocols that are compatible with LRC have been developed: the TreadMarks (Tmk) protocol [11] and the Princeton home-based (HLRC) protocol [20].

Some aspects of these protocols are the same. In particular, modifications to (shared) pages are detected and captured using *twinning* and *diffing*. Briefly, the implementation works as follows. A page is initially write protected, so that a protection fault occurs at the first write. The system then makes a copy of the page, the twin, and removes the write protection so that further writes to the page can occur without any additional overhead. The twin and the current copy are later compared to create a diff, a run-length encoded record of the modifications to the page.

The protocols differ, however, in the location where updates are kept and in the way that a processor updates its copy of a page. In Tmk, processors update a page by fetching diffs from the last writer, or writers to the page. In HLRC, every page is statically assigned a home processor by the program where writers flush their modifications at release time. To update a page a processor requests the home node for a brand new copy of the page.

2.3 Comparing Tmk and HLRC

Both protocols have strengths and weaknesses when compared to each other. For migratory data, the TreadMarks protocol uses half as many messages, because it transfers the diff(s) directly from the last writer to next writer. (The current implementation of the TreadMarks protocol avoids the problem of diff accumulation through the methods described by Amza et al. [2].) For producer/consumer data, the two protocols use roughly the same number of messages. Any difference in favor of the home-based protocol is a result of data aggregation. If a consumer is the home for several pages written by a producer, the producer can easily aggregate multiple diffs in a single message to the consumer/home. Any difference in favor of the TreadMarks protocol is typically a result of the producer and/or consumer changing, such that neither the producer nor the consumer are the home. For data in a falsely shared page, that is written by multiple processors and then read by multiple processors, the difference between the protocols is the greatest. The home-based protocol uses significantly fewer messages as the number of readers and writers increases. Specifically, for R readers and W writers, the home-based protocol uses at most $2W + 2R$ messages and the TreadMarks protocol uses at most $2WR$ messages. If, however, there are multiple writers followed by a *single* reader, then the number of messages is basically the same.

Regardless of the sharing pattern, the assignment of pages to homes is extremely important to the performance of the home-based protocol. A poor assignment can increase the number of diffs created, messages passed, and bytes transferred by an order of magnitude. On the other hand, a good assignment can have benefits that are not possible with the TreadMarks protocol. For example, in the case of a multi page data structure that is read by multiple readers following a single writer, HLRC can lower the communication load on the writer as well as achieve better communication load balancing by selecting the homes for the pages among the readers. On the other hand, the TreadMarks protocol has its share of advantages: it typically (1) transfers less data because it uses diffs to update faulting processors and (2) creates fewer diffs because their creation is delayed until they are requested by a reader.

Chapter 3

Contention, Protocol Load, and Adaptive Striping

In this chapter we introduce the concepts of contention and protocol load imbalance. We give some intuition on the characteristics of Tmk and HLRC that may lead to contention and protocol load imbalance. Finally, we introduce Adaptive Striping: an improvement to the Tmk protocol that automatically reduces protocol imbalance by redistribing updates for multi-paged data structures that are single writer and multiple reader.

3.1 Contention

We define contention as simultaneous requests on a node. In our experience contention can be attributed to limitations in the processor or the network. In the former case, the time that the node requires to process a request is longer than it takes for the next request to arrive. In the latter case, the node fails to push out responses fast enough due to limitations in the network. Most systems, under this condition, just wait for the interface to vacate enough buffer space.

We characterize contention based on the size of the set of pages referred to by the simultaneous requests. Contention is said to be single-paged, when all requests are for the same page, or multi-paged, when various pages are being requested.

Although contention happens at the node holding the updates, its effects are felt on the requestor. As most DSM systems handle requests in the same order in which they arrive, the service time for a request that arrives at a node with n outstanding request is augmented by the time required to process all preceding requests.

3.2 Protocol Load Imbalance

We refer to the work performed to propagate updates as protocol load (PL). Under TreadMarks, PL reflects time spent servicing requests. For HLRC, it also includes pushing modifications to home nodes. We then define PL imbalance as the difference in PL across the nodes of the system.

As Tmk and HLRC differ in the location where updates are kept, the protocols may have a different effect on the PL balance. To illustrate the difference, consider a multi-page data structure that has a single writer followed by multiple readers. Both Tmk and HLRC handle each page of the data structure in roughly similar ways. Each reader sends a request message to the processor holding the latest copy of the page. That processor sends back a reply message containing either the changes to the page in Tmk, or a complete copy of the page in HLRC. Where the protocols differ is in the location of the updates. In Tmk, the last writer is the source of the updates, while in HLRC the updates are kept at the home nodes. Hence, Tmk places the load of distributing multiple copies of the entire data structure on the last writer. In contrast, in HLRC, a clever home assigment may result in a more balanced distribution of the load among the nodes.

PL imbalances tend to increase with the number of nodes. Simply put, as we increase the number of readers of a page, the load on the processor servicing the update requests grows.

Finally, we have observed a relationship between PL imbalance and contention; where higher imbalance tends to translate into higher contention. This conclusion is intuitive since it is to be expected that when the proportion of updates coming out of any given node grows, the likelihood of simultaneous requests on that node would increase.

3.3 Adaptive Striping

Adaptive Striping is a simple extension of the Tmk protocol. Its goal is to automatically alleviate some protocol load imbalances. In particular, those that result from multi-paged data structures that are single writer but multiple reader.

Adaptive Striping reduces contention at the writer by evenly distributing the modifications to the other processors at the next global synchronization point. As a result, the writer's effort on behalf of each page that it off-loads is limited to constructing and sending the diff to a single processor. The processor receiving this diff is then responsible for servicing the requests from all of the consumers. Overall, neither the number of messages nor the amount of data transferred is reduced, but the average time that a reader waits for its requested diff drops.

Multiple-page data structures produced by a single processor and consumed by multiple processors are easy to recognize at a global synchronization point, such as a barrier. Specifically, each processor counts the number of pages modified by itself and the other processors that are (1) currently single-writer and (2) previously multiple-reader. If a majority of such pages come from a minority of processors, then we redistribute the modifications to those pages.

We could have, instead, initiated stripping only when all modifications, made in an interval, were performed by a single processor. This heuristic would, in fact, serve well our benchmarks. However, this simple mechanism is ineffective for more complex sharing patterns where there might be a few producers which service updates to several consumers. This is an important concern for large scale systems as it becomes clear that the contention in a 64 node cluster were 2 writers, serve each, updates to 31 other readers is, as severe as, the contention that occurs in a 32 nodes cluster, where a single writer serves 31 readers. Finally, by requiring that the pages must show multiple-reader behavior, we avoid false positives for programs that initialize large amounts of data on one processor at the beginning of a program.

Pages are assigned to processors using a weighted distribution. Specifically, we try to assign to each processor a group of pages that are expected to have the same number of read requests. In other words, one processor may be assigned fewer pages than another processor if those pages have a larger number of expected readers. Thus, the number of requests that each processor services will be roughly the same if the degree of read sharing for each page does not change. Since every processor has complete information regarding modifications at a global synchronization point, it is not necessary to communicate any information about where to find changes. They can compute it.

Chapter 4

Experimental Evaluation

In this section we first show that applications that exhibit good scalability do not suffer from increases in memory latency. We then argue that applications that scale poorly do so because of increases in memory latency that result from contention and imbalances in protocol load. Furthermore, we show that there is a relationship between protocol imbalance, contention, and memory latency, whereas higher imbalances tend to aggravate contention, which in turn increases memory latency.

4.1 Platform

Our experimental environment consist of a switched, full-duplex 100Mbps Ethernet network of thirty-two 300 MHz Pentium II-based computers. Each computer has a 512K byte secondary cache and 256M bytes of memory. All of the computers run FreeBSD 2.2.6 and communicate through UDP sockets. On this platform, the round-trip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies from 178 to 272 microseconds. The time for a 32-processor barrier is 1,333 microseconds. The time to obtain a diff varies from 313 to 1,544 microseconds, depending on the size of the diff. The time to obtain a full page is 1,308 microseconds.

4.2 Programs

We use five programs: Red-Black SOR and Gaussian Elimination are small kernels that are distributed with TreadMarks; 3D FFT is from the NAS benchmark suite [3];

Barnes-Hut is from the SPLASH benchmark suite [18]; and ILINK is a production computational genetics code [6, 13, 9].

Table 4.1 lists the problem size and the sequential execution time for each program. Sequential execution times were obtained by removing all TreadMarks calls from the programs. These times are used as the basis for computing the speedups reported in this paper.

Red-Black SOR is a method for solving partial differential equations by iterating over a two-dimensional shared array. Each processor is assigned a band of rows. Communication is limited to the elements in boundary rows. In every iteration processors update each of their elements to the average of the element's four nearest neighbors.

Gauss implements Gaussian Elimination with partial pivoting on linear equations stored in a two-dimensional shared array. The computation on the rows is distributed across the processors in a round-robin fashion in order to maintain a good load balance as elimination proceeds. In addition to the shared array for storing the coefficients of the equations, an index variable storing the pivot row number is also shared.

3D FFT solves a partial differential equation using three-dimensional forward and inverse FFT. The program has two shared data structures, an array of elements and an array of checksums. False sharing only happens on the checksum page. The computation is decomposed so that every iteration includes local computation and a global transpose.

Program	Size, Iterations	Seq. Time (sec.)
Red-Black SOR	8kx4k , 20	72.23
3D FFT	7x7x7 , 10	109.50
Gauss	4096 , 10	649.29
Barnes-Hut	65536 , 3	125.69
ILINK	CLP	276.27

Table 4.1 Program Characteristics

Barnes-Hut performs an N-body simulation using the hierarchical Barnes-Hut method. There are two shared data structures: a tree used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all the particles; and an array of particles corresponding to the leaves of the tree. Every iteration rebuilds the tree on a single node followed by a parallel force evaluation of the particles, during which most of the tree is read by all nodes. Updates to the particle array are characterized by a high degree of false sharing.

ILINK performs genetic linkage analysis. It is widely-used to locate a specific disease gene on a chromosome. ILINK implements the parallel algorithm described by Dwarkadas et al. [7]. The main shared data structure is a pool of genarrays with probabilities of specific genotypes for the individuals in a family tree. Updates to each individual's genarray are parallelized. For example, to update a parent's genarray, the master processor assigns the elements in the parent's genarray to itself and multiple slave processors in a round-robin fashion. This round-robin assignment results in a high degree of false sharing for the pages among the processors, but results in significantly better load balance. Each processor works on its share of the genarray. This requires reading the relevant elements of its spouse's and childrens' genarrays (falsely shared) from previous updates, as well as reading the summary for those updates from the master. After all of the processors complete their updates to the genarray, the master processor sums up the contributions of each of the processors and stores the summary in some portion of the genarray.

For every program, the (static) assignment strategy of pages to homes was selected through considerable experimentation. The results in this paper reflect the strategy which wields the best performance. For Red-Black SOR, 3D FFT, Barnes-Hut, and ILINK, the best assignment follows a block distribution over each of the major data structures. In other words, each of the shared data structures was divided into equal-sized groups of consecutive pages, according to the number of processors; and each processor is assigned a single group. For Gauss, the best assignment follows a cyclic (or round-robin) distribution.

4.3 Results

In this section we present the results of running the programs on 8, 16, and 32 processors. Figures 4.1 through 4.13 present the speedup and a breakdown of the execution time. Tables 4.2 through 4.6 show the variations in average response time. They also provide the average, per node, number of messages sent, the total data transfered, and the global message count. Finally, figures 4.2 through 4.15 present the response time and protocol load histograms.

The breakdown of the execution time for each program (figures 4.1b through 4.13b) has three components: `memory` is the time spent waiting to update a page; `synchro` is time waiting for synchronization to complete; and `computation` includes all other time.

The response time histograms plot the time necessary to fulfill every request sent by every node in the system. On the horizontal axis is the time in hundreds of microseconds to complete a request. On the vertical axis is the percentage of requests sent by all nodes that required a certain amount of time to be completed.

The protocol load histograms plot the time spent servicing remote requests by every node. Each bar is composed of three elements: `communication` corresponds to time spent receiving and decoding requests, as well as sending replies; `diff` corresponds to time spent building diffs; and `wait corresponds` to time spent waiting for the network interface to vacate I/O buffers. For Gauss, the protocol load histograms reflect only the time elapsed during the 8th iteration, instead of the entire execution. This finer resolution is necessary to show the imbalance in protocol load that occurs during a iteration.

Red-Black SOR is included as a control program. It is a program that achieves good scalability (with a speedup of 25.7 on 32 processors) and does not suffer from increases in response time; as can appreciated by the small memory component in figure 4.1b as well as by the minor variations in the response time histograms of figure 4.2. Furthermore, it exhibits little contention and has a good balance of protocol load; which is evident in the similar size of the bars of the histograms in figure 4.3. For the rest of this section we will use the Red-Black SOR response time and protocol load plots of figure 4.2 and figure 4.3 to illustrate how response time and protocol load histograms should look in the absence of contention and protocol load imbalances.

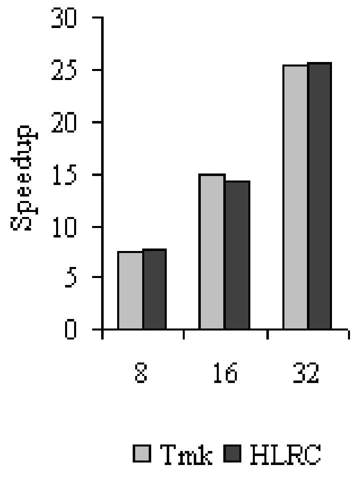
The rest of this section is divided into four parts. First, we discuss the execution time breakdowns. Second, we talk about the various types of contention exhibited by our programs and how they increase the latency of individual requests. Third, we present our results on protocol load imbalance. Finally, we quantify the effect that contention has on the programs' speedups by eliminating contention manually or estimating its cost analytically.

4.3.1 Execution Time Breakdown

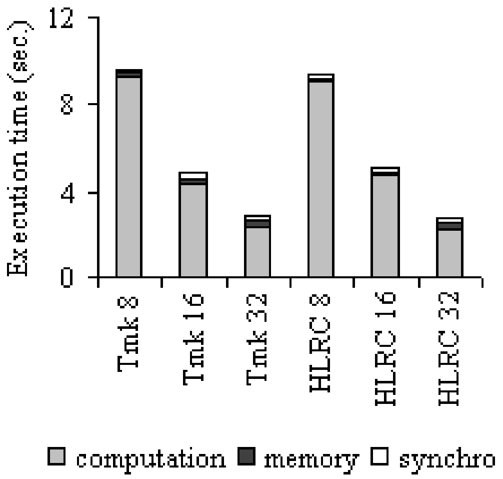
All of the programs, except for SOR, exhibited significant increases in access miss time as the number of processors increased. Gauss was affected the most. As figure 4.7b

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	1592.33	100	5.6	2135
	16	1625.89	116	12.0	4879
	32	1668.53	125	27.0	10367
HLRC	8	1400.43	83	6.7	2135
	16	1409.80	99	15.0	4878
	32	1474.77	106	34.0	10366

Table 4.2 Response time is almost unchanged for Red-Black SOR. The table shows the average response time, average number of requests per node, total data transfered and total message count of Tmk and HLRC.



(a) Speedup



(b) Execution time breakdown

Figure 4.1 Scalability of Tmk and HLRC for Red-Black SOR.

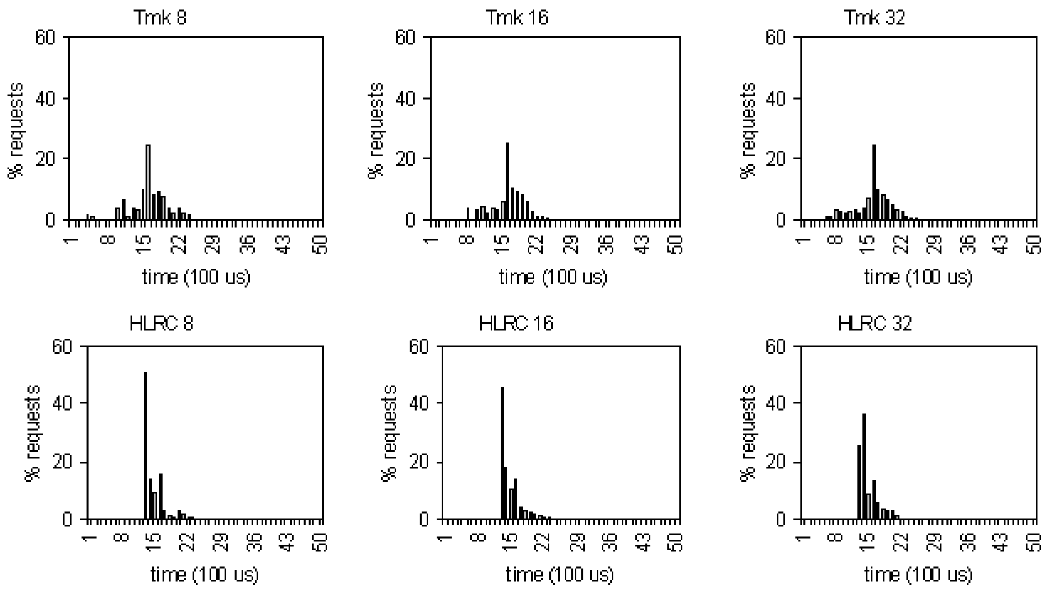


Figure 4.2 Response time histograms of Tmk and HLRC for Red-Black SOR. The small variation in response time is evident in the mostly unchanged Tmk and HLRC plots.

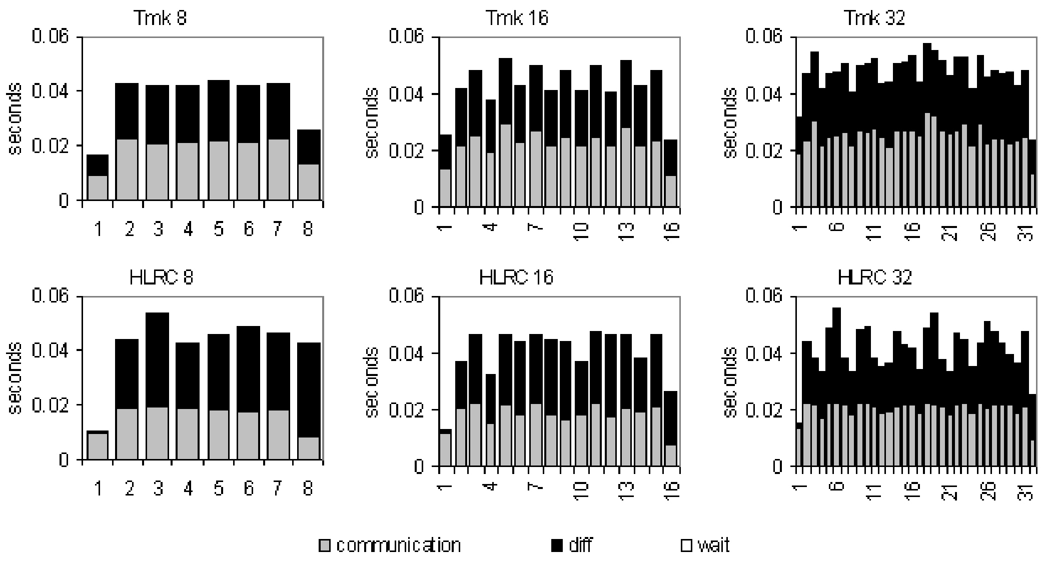
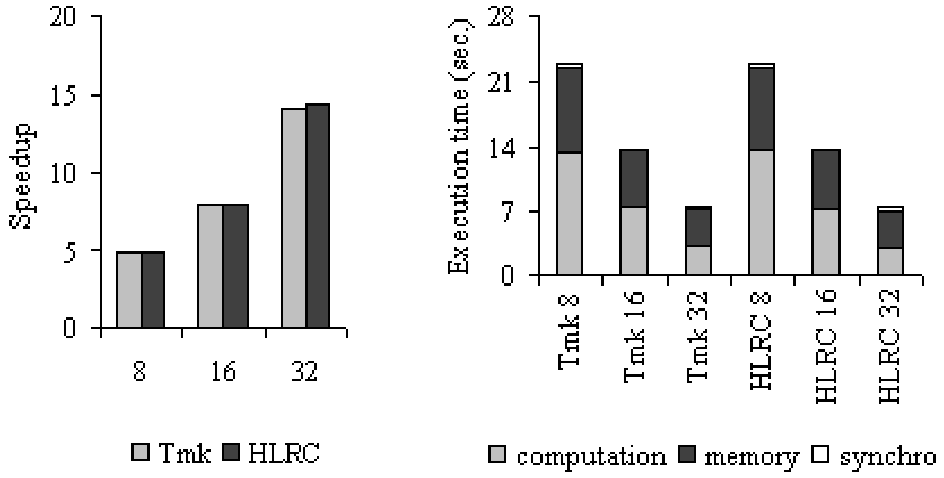


Figure 4.3 Protocol load histograms of Tmk and HLRC for Red-Black SOR. The Tmk and HLRC graphs show that protocol load is balanced across nodes.



(a) Speedup (b) Execution time breakdown

Figure 4.4 Scalability of Tmk and HLRC for 3D FFT.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	2017.00	4041	265	65785
	16	2449.40	2169	284	74003
	32	2963.19	1125	295	90514
HLRC	8	1988.77	4041	265	65093
	16	2412.35	2169	285	70365
	32	2918.72	1125	297	73997

Table 4.3 Response time increases for 3D FFT. Response time increased by 47% for both Tmk and HLRC. The table shows the average response time, average number of requests per node, total data transfered and total message count of Tmk and HLRC.

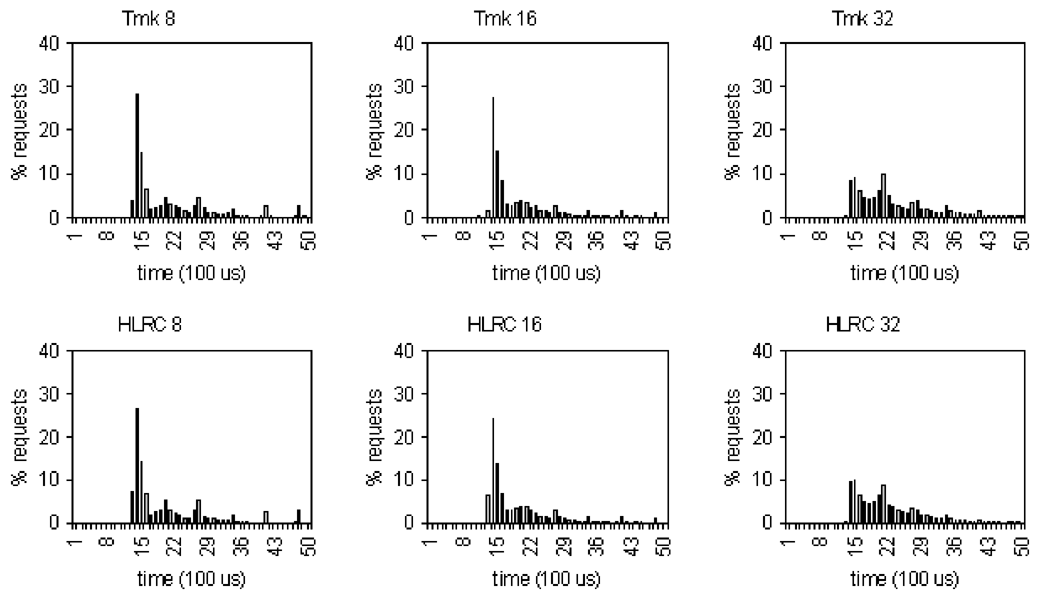


Figure 4.5 Response time histograms of Tmk and HLRC for 3D FFT. The Tmk and HLRC graphs show a significant degradation of response time. The higher latency leads to plots with shorter bars that are shifted to the right and spread out.

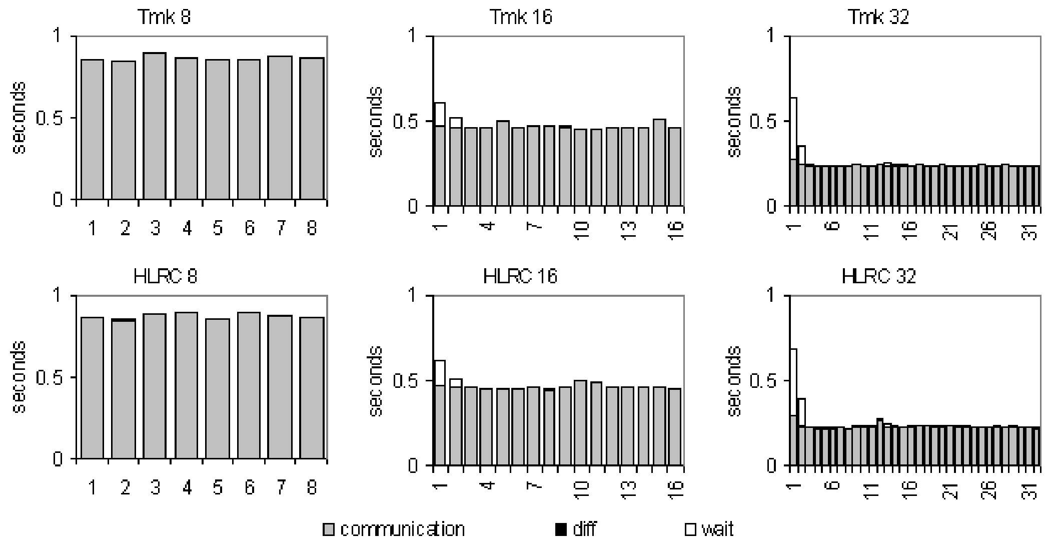


Figure 4.6 Protocol load histograms of Tmk and HLRC for 3D FFT. The high number of simultaneous requests is evident in the large wait component of the left most processors.

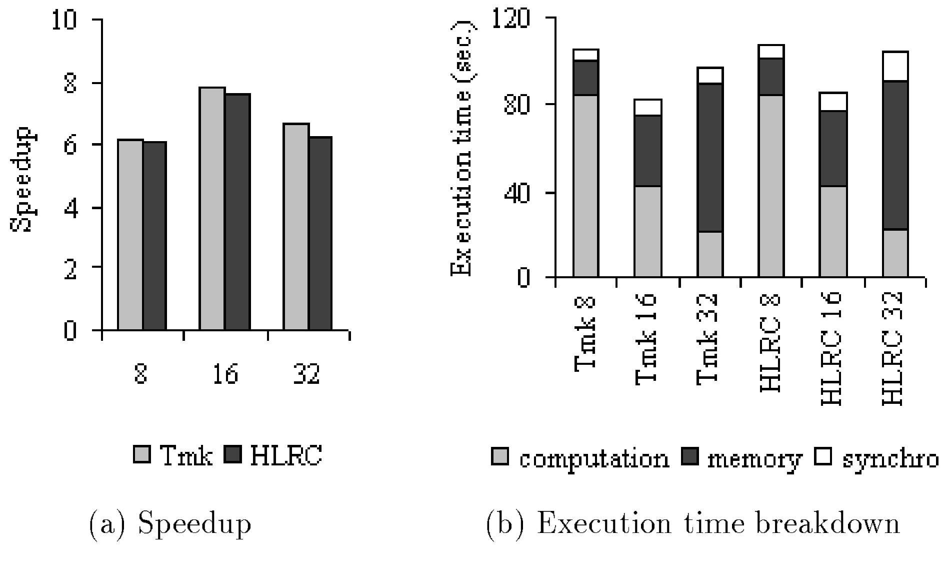


Figure 4.7 Scalability of Tmk and HLRC for Gaussian Elimination.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	2595.65	8065	357	200683
	16	4848.49	8441	766	429915
	32	8954.12	8929	1581	887995
HLRC	8	2882.89	8065	562	200669
	16	4909.63	8441	1236	429882
	32	8640.11	8929	2586	887922

Table 4.4 Response time increases for Gauss. Response time increases by 245% for Tmk and by 200% for HLRC. The table shows the average response time, average number of requests per node, total data transfered and total message count of Tmk and HLRC.

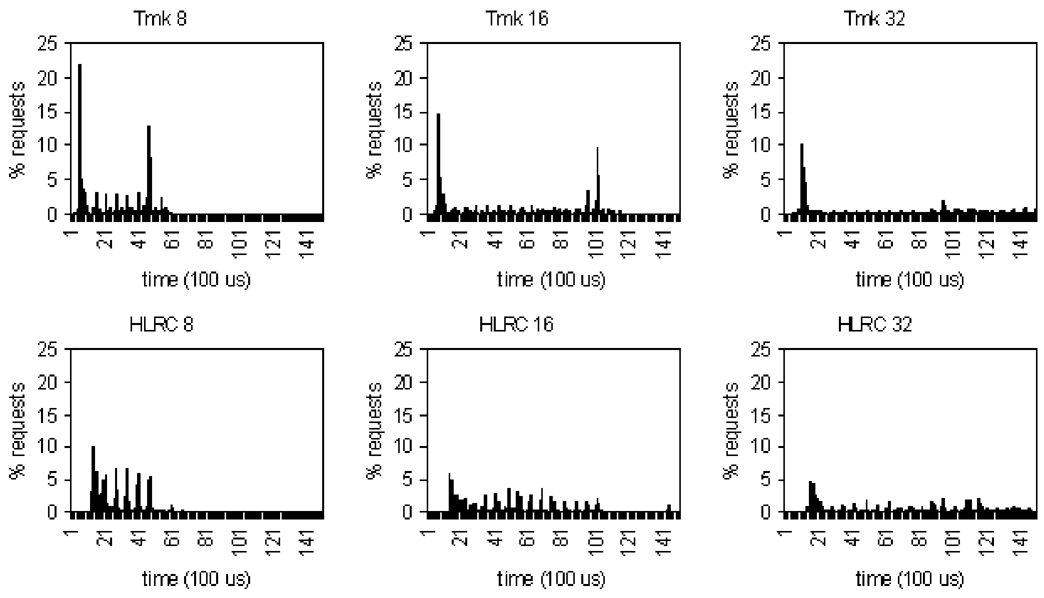


Figure 4.8 Response time histograms of Tmk and HLRC for Gauss. The Tmk and HLRC graphs show a significant degradation of response time. The higher latency leads to plots with shorter bars that are shifted to the right and spread out.

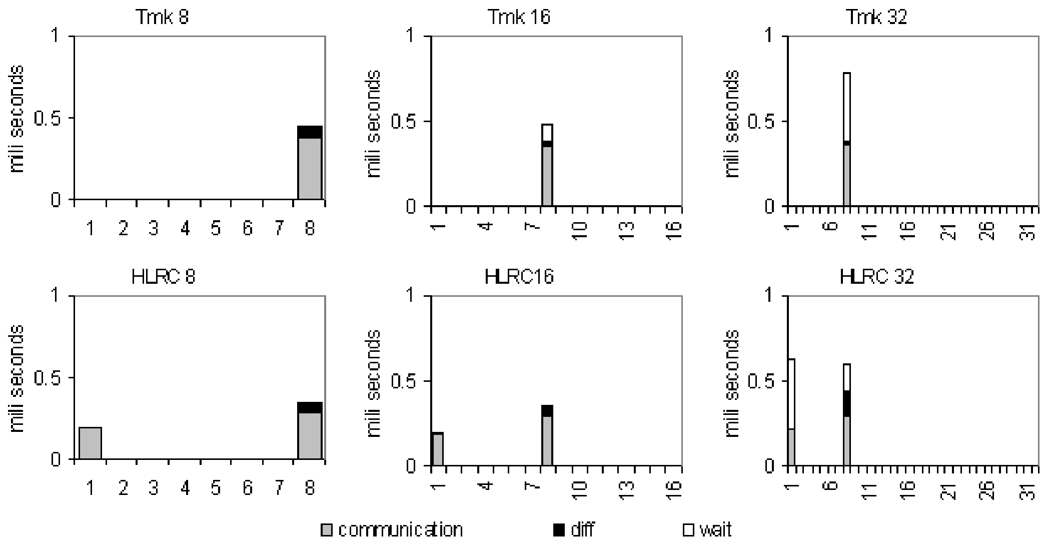
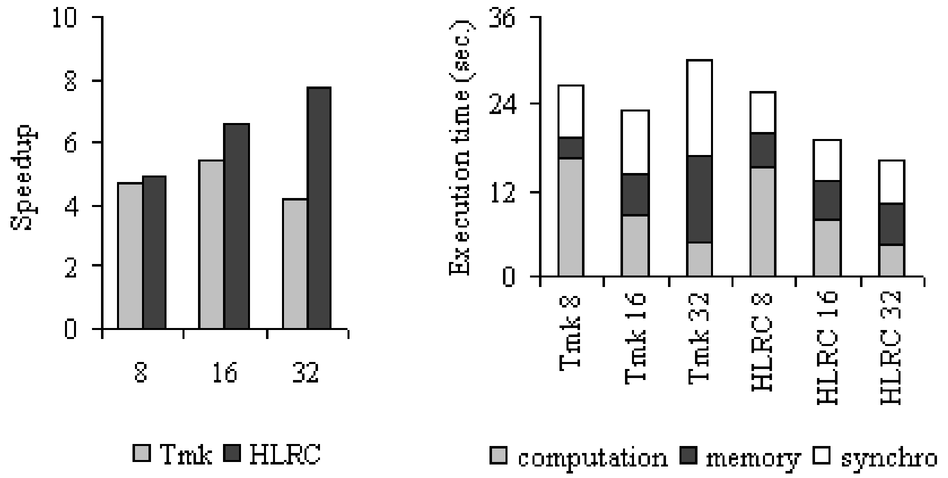


Figure 4.9 Protocol load histograms of Tmk and HRLC for Gauss. The graphs plot the protocol load for the 8th iteration. For Tmk all protocol load is concentrated in processor 8, the owner of the pivot row. For HLRC load is divided among procesor 0, home for the pivotal index, and processor 8, the owner of the pivot row.



(a) Speedup (b) Execution time breakdown

Figure 4.10 Scalability of Tmk and HLRC for Barnes-Hut.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	1630.94	2442	130	144038
	16	2756.69	2154	247	441679
	32	5534.01	2026	488	1535326
HLRC	8	1655.81	2072	154	34306
	16	1825.95	1964	278	64726
	32	2033.12	1930	529	128790

Table 4.5 Response time increases for Barnes-Hut. There is a significant difference between the response time increase of Tmk and HLRC (239% vs. 23%). The table shows the average response time, average number of requests per node, total data transfered and total message count of Tmk and HLRC.

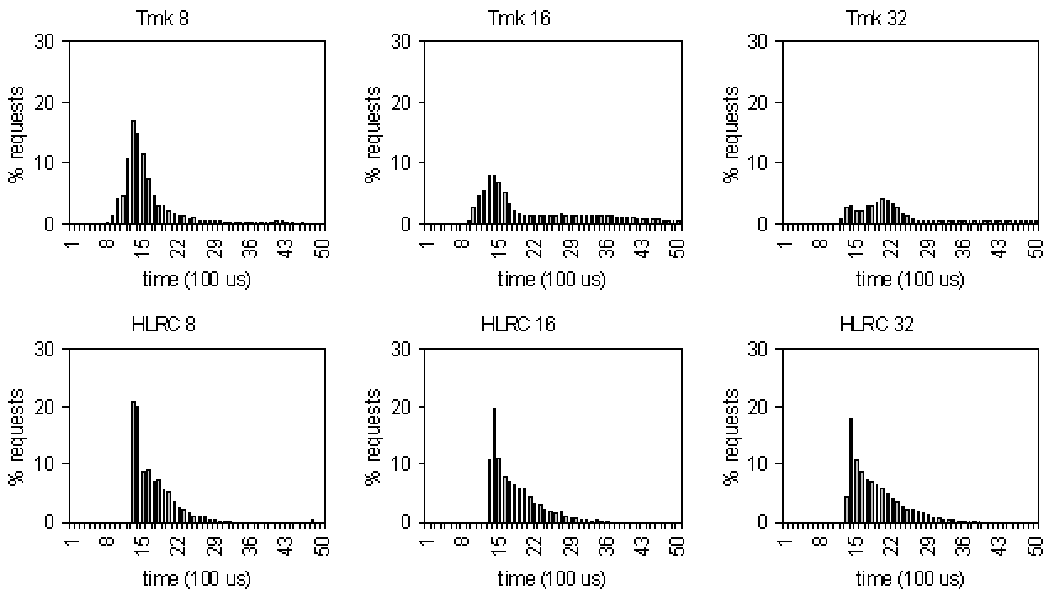


Figure 4.11 Response time histograms of Tmk and HLRC for Barnes-Hut. The Tmk plots show a siginifican degradation of response time, while HLRC plots show little change. The higher latency in Tmk leads to plots with shorter bars that are shifted to the right and spread out.

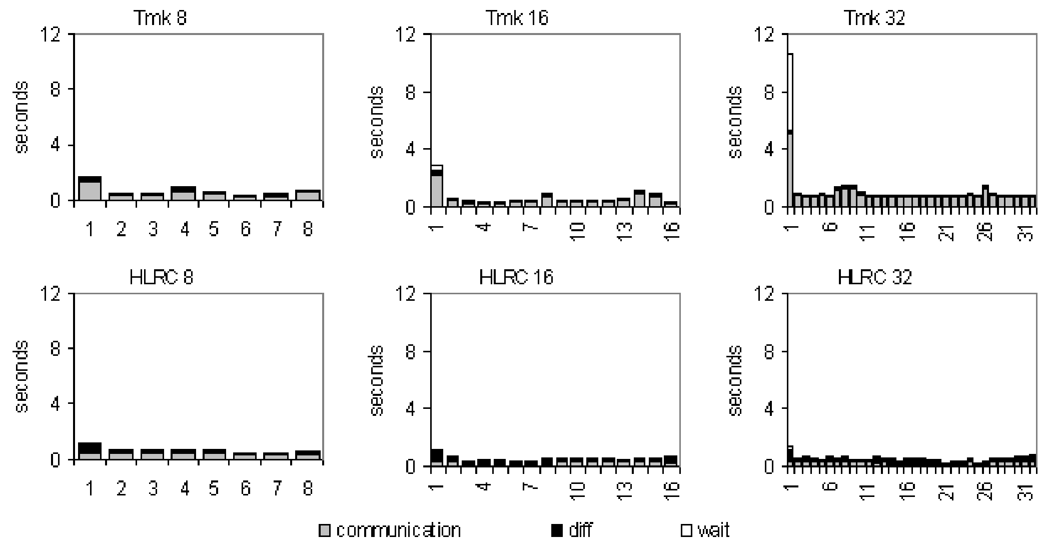


Figure 4.12 Protocol load histograms of Tmk and HLRC for Barnes-Hut. The high load imbalance in Tmk is attributed to the distribution of updates for the tree. On the 32 node cluster, protocol load accounts for 35.6% of total execution time of Tmk. Of that 50.42% is spent blocked waiting for the network interface to clear space in the I/O buffers. HLRC's protocol load is balanced.

shows, on 32 processors, under any of the protocols, access miss time accounts for 65% of the execution time.

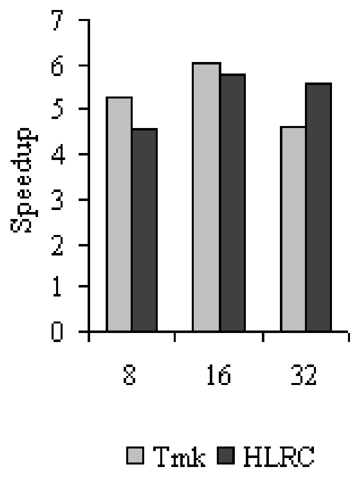
4.3.2 Contention

We argue that the increase in access miss time experienced by our programs is not solely the result of an increased number of request messages. It is instead largely a result of the increase in latency for individual requests due to contention.

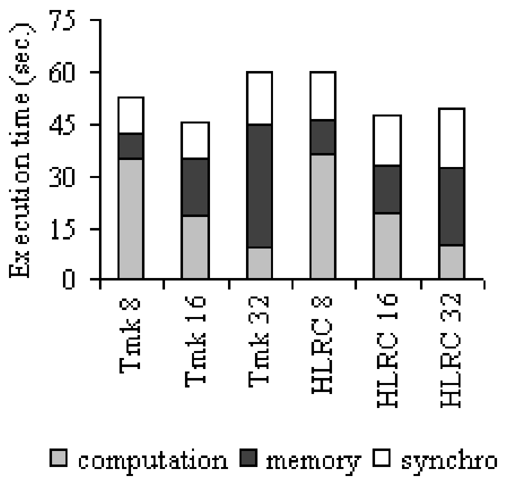
This trend is most evident for 3D FFT and Gauss. In 3D FFT, the average number of requests per node drops significantly as the size of the cluster increases. In Gauss, it increases moderately (4.7%). Both programs, however, experience sharp increases in response time. The average request latency increases for 3D FFT and Gauss by 46% and 245%, respectively (tables 4.3 and 4.4). The increase in request latency is also evident in the response time histograms (figures 4.5 and 4.8). The higher latency leads to plots with shorter bars that are shifted to the right and spread out. These plots stand in sharp contrast with the response time histograms for SOR (figure 4.2) where the plots remain mostly unchanged.

3D FFT suffers from multi-page contention. In 3D FFT, processors work on a contiguous band of elements from the shared array. Since HLRC home assignment is done in blocks, in both protocols the processor modifying the band will be the one hosting the update. The computation on the 3D arrays is partitioned along one axis among the processors. As a result, the global transpose leads to all processors trying to read some pages from processor 0 first, and then some pages from processor 1, and so on, resulting in a temporal contention at a single processor at a time. The effects of contention are visible in the plots of figure 4.6.

The Tmk and HLRC plots show significant higher protocol loads for the two leftmost processors, in particular for the 16 and 32 node clusters. The protocol load imbalance is somehow surprising, since during the course of an iteration, all nodes service the same number of request. The explanation to this discrepancy is found in



(a) Speedup



(b) Execution time breakdown

Figure 4.13 Scalability of Tmk and HLRC for ILINK.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	1339.12	5598	105	124079
	16	2577.59	5944	232	294347
	32	5194.03	6131	513	684343
HLRC	8	2086.66	4879	346	110160
	16	2475.15	5525	757	244063
	32	3072.90	5904	1595	505418

Table 4.6 Response time increases for ILINK. There is a significant difference between the response time increase of Tmk and HLRC (287% vs. 47%). The table shows the average response time, average number of requests per node, total data transfered and total message count of Tmk and HLRC.

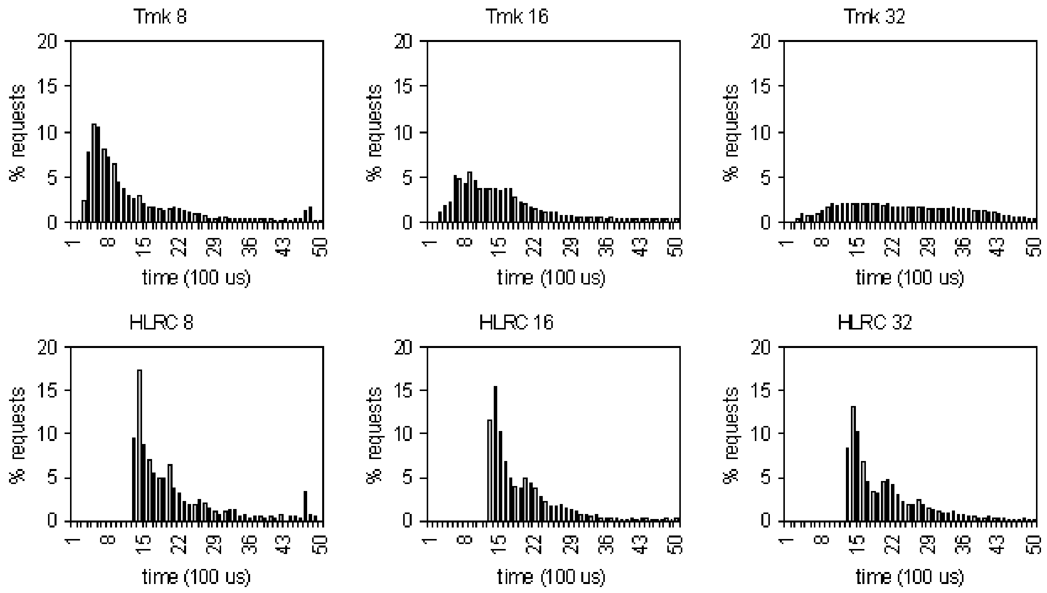


Figure 4.14 Response time histograms of Tmk and HLRC for ILINK. The Tmk plots show a significan degradation of response time, while HLRC plots show little change. The higher latency in Tmk leads to plots with shorter bars that are shifted to the right and spread out.

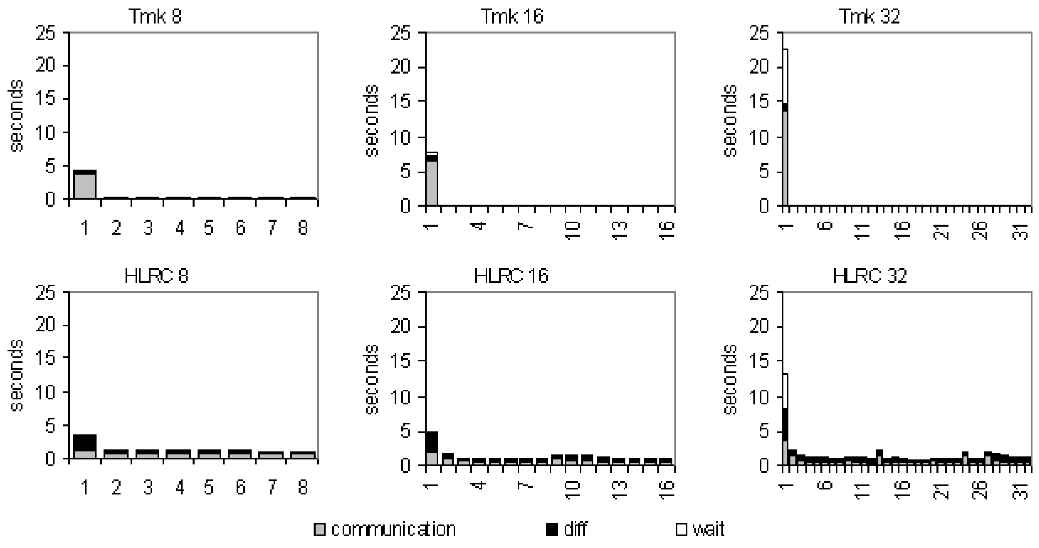


Figure 4.15 Protocol load histograms of Tmk and HLRC for ILINK. The high load imbalance in Tmk is attributed to the distribution of updates for the summaries. On the 32 node cluster, protocol load accounts for 37.9% of total execution time of Tmk. Of that 35.11% is spent blocked waiting for the network interface to clear space in the I/O buffers. The protocol load imbalance in HLRC is attributed to the changing access pattern of the application's, as well as, the placement, on processor 0, of shared structures that are smaller than a page.

the order in which data is access. We mention earlier, that at the start of an iteration, all processors, at once, try to fetch data from processor 0. The simultaneous request create contention on this processor, as is evidenced by the high wait component in the plots. However, as processor 0 services these request, it implicitly orders communication, distributing replies over time. The distribution of replies, ensures that the next round of communication will see requests that are more spread out, producing less contention, as is evidenced by the smaller wait component of the next processor. Processor 1, in turn, will further distribute replies, in effect slowing down some nodes, and further limiting the number of simultaneous requests on the latter accesses.

Gauss suffers from single-page contention. The access pattern for the pivot row is single-writer/multiple-reader. There are only two such pages for our problem size, but their producer changes on every iteration. The severity of contention in Gauss is shown in the plots of figure 4.9. The large wait component in the Tmk and HLRC plots results from the network interface inability to service requests, for the pivot row and index, as fast as they arrive. The protocol load in the Tmk plot is concentrated in processor 8, the last writer to the pivot row and pivot index. For HLRC, the load is divided among processor 0, home for the index, and processor 8 home for the pivot row.

Both Barnes-Hut and ILINK suffer from single and multi-page contention. The tree and array of particle in Barnes, and the genarrays in ILINK suffer from single-page contention when read in the same order by multiple nodes (i.e., the tree root for Barnes, and the summaries for ILINK). Additionally, multi-page contention occurs when updates for multiple parts of the data structures reside at any given node. For both Barnes and ILINK there is a significant difference in the rate of latency increase between Tmk and HLRC (tables 4.5 and 4.6). While the average response time for HLRC increases slowly, it almost doubles with an increase in the number of nodes for Tmk. This is supported by the rapid dispersion of the bars in the Tmk histograms (figures 4.11 and 4.14). We argue in the next section that differences in protocol load balance, shown in figures 4.12 and 4.15, account for the disparity in response time.

4.3.3 Protocol Load Imbalance

We argue that imbalances in protocol load have the potential to aggravate contention and increase request latency. Furthermore, we argue that protocol load imbalance accounts for the difference in request latency experienced between Tmk and HLRC for Barnes-Hut and ILINK.

We prove our claim by presenting results for Adaptive Striping. Adaptive Striping is an improvement to the Tmk protocol that automatically reduces contention by eliminating protocol imbalances created by multi-page data structures with a single writer and multiple readers.

Figures 4.16 and 4.19 compare the speedup and breakdown of execution time of Tmk, HLRC and Tmk with Adaptive Striping (Tmk/stripe) for Barnes-Hut and ILINK. Tables 4.7 and 4.8 show the variations in average response time. They also provide the average, per node, of the number of messages sent, the total data transfered, and the global message count of Tmk/stripe for Barnes-Hut and ILINK. Finally, figures 4.17 through 4.21 present the response time and protocol load histograms of Tmk/stripe for Barnes-Hut and ILINK.

These results show that once the protocol imbalance is removed (compare figures 4.12 and 4.15 to figures 4.18 and 4.21), Tmk/stripe achieves speedups for Barnes-Hut and ILINK that are comparable to and 46% higher than HLRC (figure 4.16 and 4.19), in spite of sending 12 and 1.35 times more messages. The difference in the number of messages results from the different ways that the protocols handle false sharing. All versions send about the same amount of data.

The high load imbalance in Tmk is attributed to the distribution of the updates for the tree in Barnes, and the summaries in ILINK. In Tmk updates are always fetched from the last writer, hence processor 0 has to supply all updates to the tree in Barnes-

Hut and the summary data structures in ILINK. As processor 0 gets overwhelmed by requests for updates to the tree, it has to spend an increasing portion of its execution time servicing requests. Figures 4.12 and 4.15 show that on the 32 node cluster this time accounted for 35.6% and 37.9% of the total execution time. Of that time, 50.42% and 35.11% was spent blocked waiting for the network interface to clear space in the I/O buffers.

In contrast HLRC greatly alleviates the contention for reading the tree and summaries by assigning the homes for these data structures across the processors. Specifically, if the tree covers n pages and every processor reads the whole tree, then Tmk requires processor 0 to service *(p-1)*n* page requests. HLRC instead distributes the tree in *n*(p-1)/p* messages. After that the load of servicing the tree requests is evenly distributed.

In Barnes-Hut, adaptive striping eliminates protocol load imbalance caused by requests from multiple processors to processor 0 in order to obtain the tree data. In ILINK, it reduces contention for reading the summary part of the genarrays. For Barnes-Hut, Tmk/stripe achieves comparable protocol load imbalances as HLRC. For ILINK, Tmk/stripe achieves four times lower protocol load on processor 0 than HLRC.

When comparing the response time histograms for Barnes-Hut and ILINK under Tmk/stripe (figures 4.17 and 4.20), we notice that they resemble the graphs for HLRC (figures 4.11 and 4.14). For Barnes-Hut, Tmk/stripe achieves average response times comparable to HLRC. For ILINK, Tmk/stripe response time is 28% lower than HLRC. The lower response time results from Tmk/stripe sending diffs instead of full pages, as well as from its better balance of protocol load.

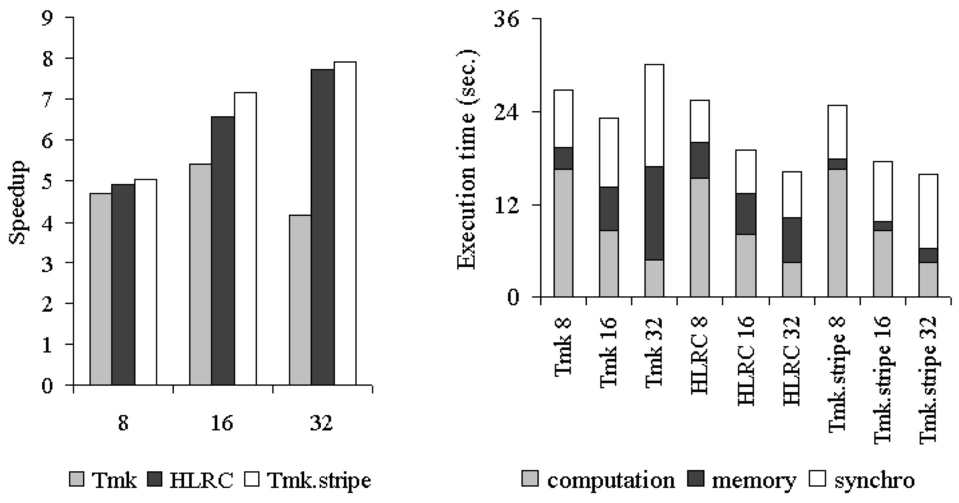
These graphs demostrate that (for these applications, at least) there is a relationship between protocol balance and response time. It is interesting to note that as protocol imbalance grows, so does response time. This is an expected result, since it follows that when the proportion of updates coming out of any given node grows, the likelihood of simultaneous resquests (i.e. contention) on that node would increase.

ILINK illustrates why dynamic striping can be better than the static striping that is possible with HLRC. In ILINK, the multiple-reader pages fall into several groups of consecutive pages. Each group of consecutive pages represents a single individual. In the computation of each nuclear family, different groups are used. To optimally stripe for the computation of a nuclear family, you must adapt the striping for the groups used in that computation. With a single static striping over all of the groups in the genarray, a particular nuclear family computation may use a collection of groups that are not evenly striped. In other words, because of the combination of groups used in that nuclear family computation, some processors service more pages than others. For ILINK, Adaptive Striping does better than any static striping because it performs the striping at the barrier immediately after the initialization by processor 0 that determines which groups will be used for that nuclear family computation. So, it always evenly stripes the pages.

Nonetheless, Tmk/stripe and HLRC still suffer from contention, as shown by the difference in average response time when compared to the control (1734.28 for Barnes and 2192.22 for ILINK vs. 1474.77 for SOR). As some parts of the shared data structures are traversed in the same order, and various parts reside in the same node, simultaneous requests for updates may still reach the nodes. That is, the distribution of updates to various processors does not eliminate contention for the data, it only makes it less frequent on a particular node.

4.3.4 Quantifying the Effects of Contention on Speedup

In this section, we quantify the effects of contention on the execution time of our program suite. For Barnes-Hut, ILINK, and Gauss, we estimate the effects analytically. Our estimate is really an upper bound on the possible speedup without contention.



(a) Speedup (b) Execution time breakdown

Figure 4.16 Scalability comparison of Tmk, HLRC, and Tmk with Adaptive Stiping for Barnes-Hut.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	1510.39	2442	130	144038
	16	1587.22	2154	247	441679
	32	1734.28	2026	448	1535326

Table 4.7 Response time increases of Tmk with Adaptive Stiping for Barnes-Hut. The table shows the average response time, average number of requests per node, total data transfered and total message count.

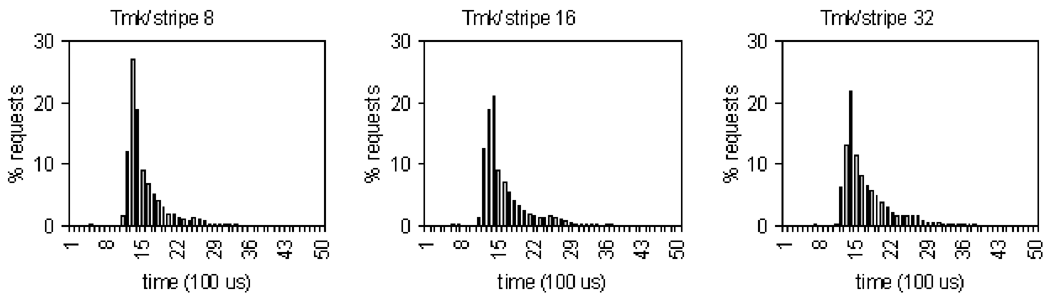


Figure 4.17 Response time histograms of Tmk with Adaptive Striping for Barnes-Hut. The plots show little change in response time and are very similar to the HLRC's plots of figure 4.11.

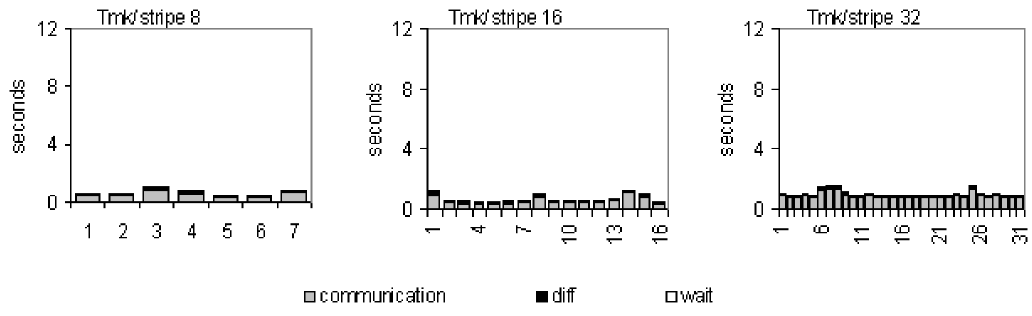
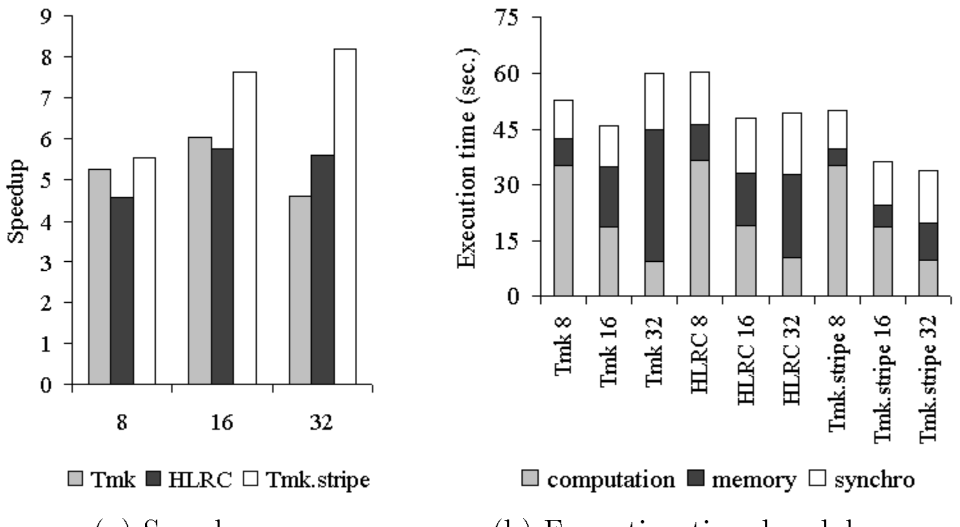


Figure 4.18 Protocol load histograms of Tmk with Adaptive Striping for Barnes-Hut. The imbalance has been removed and the plots are very similar to the HLRC's plots of figure 4.12.



(a) Speedup (b) Execution time breakdown

Figure 4.19 Scalability comparison of Tmk, HLRC, and Tmk with Adaptive Stiping for ILINK.

	Nodes	Avg. resp. time (sec.)	Avg. per node requests	Data (Mbytes)	Messages
Tmk	8	999.49	5598	105	124079
	16	1339.23	5944	232	294347
	32	2192.22	6131	513	684343

Table 4.8 Response time increases of Tmk with Adapative Stiping for ILINK. The table shows the average response time, average number of requests per node, total data transfered and total message count.

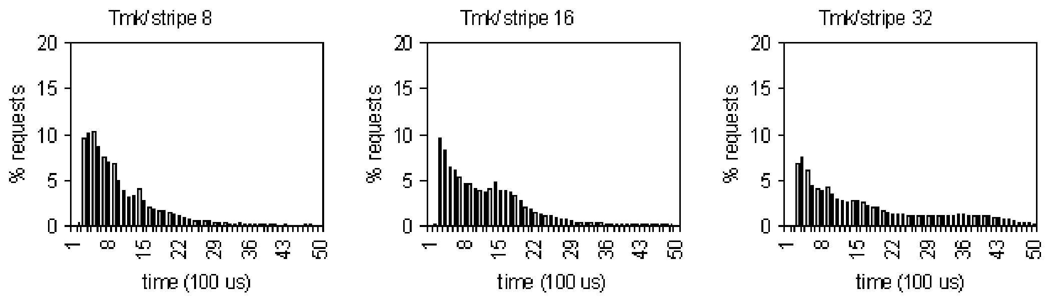


Figure 4.20 Response time histograms of Tmk with Adaptive Stripe for ILINK. The plots show little change in response time and are very similar to the HLRC's plots of figure 4.14.

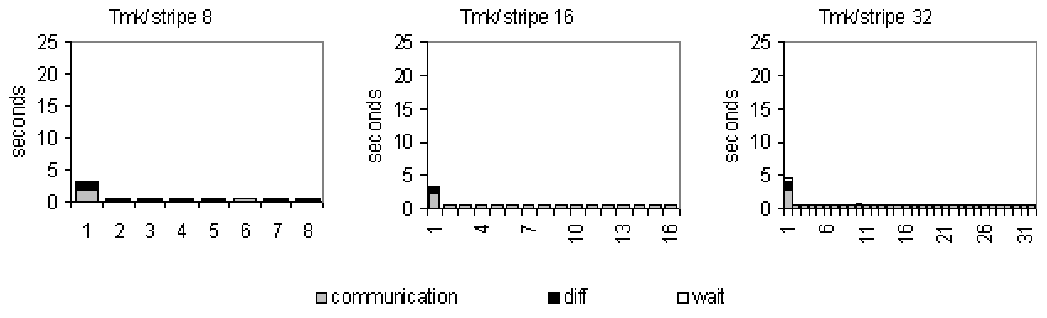


Figure 4.21 Protocol load histograms of Tmk with Adapative Striping for ILINK. The imbalance has been removed and the plots are very similar to the HLRC's plots of figure 4.15.

For 3D FFT, we modified the program to eliminate contention and measured the results.

Table 4.9 summarizes the results for Barnes-Hut, ILINK, and Gauss. We limited our analysis to HLRC because its use of full pages simplifies the analysis. (In contrast, under Tmk, we would have to account for the variable size of the diffs.) We use the execution time breakdown for the node that arrived last at the barrier most often. In effect, this node is delaying the others, limiting the speedup. To estimate speedup in the absence of contention, we first calculate the memory access time in the absence of contention. This was done by multiplying the total number of page requests for the node by the minimal page request latency (1308 microseconds). The contention free execution time was then calculated by adding the synchro, computation and contention free memory components of execution. The contention free speedup was computed from the contention free execution time and the sequential execution time.

The contention in 3D FFT, however, can be eliminated by carefully restructuring the transpose loop. By staggering the remote accesses of different consumers such that they access pages on different producers in parallel, the speedup of 3D FFT is improved from 14.14 to 16.74 on 32 PCs using Tmk.

	Time (sec.)			# of page		Contention Free	
	Synchro	Memory	Comp.	requests	Speedup	Memory	Speedup
Barnes-Hut	5.92	5.73	4.59	3068	7.73	4.01	8.65
ILINK	16.63	23.72	9.10	6080	5.59	7.95	8.20
Gauss	1.34	68.32	34.76	8929	6.22	11.68	13.59

Table 4.9 Quantifying the effects of contention on the speedup of Barnes-Hut, ILINK, and Gauss on 32 nodes. "Synchro", "Memory" and "Computation" are taken from the execution time breakdowns for HLRC on 32 nodes. "Contention Free Memory" and "Contention Free Speedup" are computed estimates.

Chapter 5

Related Work

A large number of software shared memory systems have been built. Many of the papers looking at the performance of software DSM on thirty-two or more processors have used SMP-based nodes [17, 19, 16]. Thus, the actual number of nodes on the network is typically a factor of two to eight less than the number of processors. Because requests for the same page from multiple processors within a node are combined into one, the load on the processor(s) servicing the page may not be as high as when the number of nodes in the network equals the number of processors. These studies have ignored the effects of contention and protocol load imbalance, as these become significant only on a network with a large number of nodes.

Two papers that look at large networks of uniprocessors are Zhou et al. [20] and Bal et al. [4]. Zhou et al. evaluated the home-based lazy release consistency protocol against the basic LRC protocol on an Intel Paragon. The relatively large message latency, page fault, and interrupt times compared with memory and network bandwidth, and the extremely high cost of diff creation on the Paragon architecture are uncommon in modern parallel platforms and are biased towards the HLRC protocol. The high diff cost, led the authors to conclude that the performance gap between Tmk and HLRC results from the bast differences in message count. We show that the performance gap results, instead, from the difference in protocol load balance.

Bal et al. evaluated Orca, an object-based distributed shared memory system, on a Myrinet and a Fast Ethernet connected of 32 200MHz Pentium Pro computers. The object-based DSM system decreases the number of messages and data from reduced false sharing at the cost of the programmer's extra effort to explicitly associate shared data structures with objects. Objects with low read/write ratio are stored in a single processor, while those with high read/write ratio are replicated on all processors using multicast.

Monnerat and Bianchini [15] uses a method similar to Adaptive Striping to update consumers of producer/consumer pages after a barrier. It differs however in that it updates all consumers instead of just one as is the case for adaptive striping. Moreover, their objective was to reduce memory latency by selective update. In contrast we aim to reduce the total bandwith requirement from any given node.

Chapter 6

Conclusions

We evaluate the performance of three page-based, multiple-writer protocols implementing Lazy Release Consistency (LRC) [11] on a network of 32 uniprocessors. We use two existing protocols, Princeton's home-based protocol [20] and the TreadMarks protocol [12], and a third novel protocol, Adaptive Striping. Adaptive Striping is a simple extension of the TreadMarks protocol. Its goal is to eliminate load imbalance (within the protocol) at the nodes automatically.

In general, we show that memory latency increases due to contention and protocol load imbalances are a significant obstacle to scalability. We argue that contention reduction and protocol load balancing should be considered, in addition to message reduction, by designers of scalable DSM systems.

The results show that programs that exhibit good scalability experience minor variations in memory latency (for example, only a 4.7% increase as we quadruple the number of nodes), do not suffer from contention, and have a good balancing of protocol load. In contrast, programs that scaled poorly suffered from large memory latency increases (up to 280% higher) due to contention and protocol load imbalances. Furthermore, the results show a relationship between contention and protocol load balancing: Higher protocol load imbalances usually results in increased contention. Intuitively, an increase in the proportion of data distributed from a node, increases the likelihood of simultaneous requests to that node.

For most of our programs, Tmk and HLRC are equally affected by contention and achieve similar performance. Where they differ significantly, the protocol load imbalance (caused by the read accesses to pages that have multiple readers following one or more writers) is the largest factor accounting for the difference.

We described and evaluated Adaptive Striping. It is an extension to the TreadMarks protocol that automatically alleviates contention by reducing protocol load imbalance produced by multi-page data structures that are written by one node but read by multiple nodes. In our evaluation, we found that Adaptive Striping achieves performance equal to or better than the best of the TreadMarks or the best tuned home-based implementation without requiring the manual assignment of homes.

Furthermore, the results for Adaptive Striping show that the distribution of the messages is no less important than the number of messages. For example, for Barnes-Hut, Adaptive Striping and the home-based protocol transfer roughly equal amounts of data and have identical message distributions. Although Adaptive Striping sends 12 times more messages than the home-based protocol, they achieve the same speedup.

Contention still, however, occurs in both Adaptive Striping and HLRC. The striping of updates for multi-page data structures that suffer from single-page contention, whether it is by static home assignment as in HLRC or by dynamic placement as in Adaptive Striping, does not eliminate contention for the data. It only makes contention less frequent on a particular node.

There are still sources of contention that our protocol extensions do not address. In 3D FFT, the contention was caused by multiple processors accessing different single-writer/single-reader pages at the same time. By manually restructuring the transpose loop, we found that the speedup could be improved from 14.14 to 16.74 on 32 processors. In Gauss, the contention is due to a single-writer/multiple-reader sharing pattern, but two factors prevent our protocol extensions from helping. One is that only two or three pages are read at a time, so Adaptive Striping can not help much. The other is that each page is only read by each processor (other than its producer) once. So, there is not a repeated pattern to trigger Adaptive Striping.

Bibliography

[1] S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 261–271, February 1997.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[4] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek. Performance evaluation of the Orca shared object system. *ACM Transactions on Computer Systems*, 16(1), February 1998.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[6] R.W. Cottingham Jr., R.M. Idury, and A.A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[7] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A. L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[9] S.K. Gupta, A.A. Schäffer, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Integrating parallelization strategies for linkage analysis. *Computers and Biomedical Research*, 28:116–139, June 1995.

[10] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, October 1995.

[11] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[12] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[13] G.M. Lathrop, J.M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proceedings of National Academy of Science, U.S.A.*, 81:3443–3446, June 1984.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] L.R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[16] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh. Home-based SVM protocols for SMP clusters: design and performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[17] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[18] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[19] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[20] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.