

Efficiently Adapting to Sharing Patterns in Software DSMs *

Luiz Rodolpho Monnerat[‡] and Ricardo Bianchini

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil

[‡]E&P-Gerex/Getinf/Geinf
Petrobras
Rio de Janeiro, Brazil

{monnerat,ricardo}@cos.ufrj.br

Abstract

In this paper we introduce a page-based Lazy Release Consistency protocol called ADSM that constantly and efficiently adapts to the applications' sharing patterns. Adaptation in ADSM is based on our dynamic categorization of the type of sharing experienced by each page. Pages can be categorized as falsely-shared, migratory, or producer/consumer(s). Migratory and producer/consumer(s) pages are managed in single-writer mode, while falsely-shared data are managed in multiple-writer mode. Coherence is kept with invalidations for most types of the shared data, but updates are used for lock-protected data in migratory state and barrier-protected data in producer/consumer(s) state. We performed experiments with 6 parallel applications on an 8-node SP2 system, comparing our protocol against standard TreadMarks and a version of TreadMarks that also adapts to sharing patterns. Our results show that ADSM consistently outperforms its competitors; our protocol can improve the TreadMarks speedups by as much as 155%, while surpassing the performance of the adaptive TreadMarks implementation by as much as 67%. Our main conclusions are that our categorization and adaptation strategies are useful techniques for improving the performance of page-based software DSMs, while ADSM is a highly-efficient option for low-cost parallel computing.

1 Introduction

Software distributed shared-memory systems (DSMs) provide programmers with the illusion of shared memory on top of message-passing hardware. These systems provide a low-cost alternative for shared-memory computing,

since they can be built with standard workstations and operating systems. However, many applications running on software DSMs suffer high communication and coherence-induced overheads that limit performance.

Software DSMs based on relaxed consistency models can reduce these overheads by delaying and/or restricting communication and coherence transactions as much as possible. Relaxed consistency-based multiple-writer DSMs attempt to reduce communication and coherence overheads further by allowing two or more processors to modify their local copies of shared data concurrently and merging modifications at synchronization operations. This characteristic alleviates the effect of false sharing in systems with large coherence units.

Multiple-writer protocols do have their shortcomings, however. The overhead of detecting, recording, and merging changes to shared data is always incurred, independently of whether false sharing is indeed a problem. This overhead can be eliminated for pages that are not subject to false sharing by allowing only a single writer at a time for these pages.

A related but orthogonal issue in the design of software DSMs is the type of protocol used to keep shared data coherent. The two most common approaches here are invalidate and update-based coherence protocols. Update-based protocols usually transfer more data than necessary, but can improve performance with respect to invalidate-based protocols when the sharing patterns are such that most updates are indeed useful. More specifically, migratory data accessed inside critical sections and producer/consumer(s) data can benefit greatly from some form of update-based protocol.

Given that applications can potentially benefit from both types of coherence maintenance strategies and both multiple and single-writer modes, techniques that adapt the system behavior according to sharing patterns seem imperative. Thus, in this paper we introduce a page-based Lazy

*This work was supported in part by FINEP/Brazil grant no. 56/94/0399/00 and CNPq/Brazil.

Release Consistency (LRC) protocol called ADSM (Adaptive DSM) that constantly and efficiently adapts to the applications' sharing patterns. Adaptation in ADSM is based on our dynamic categorization of the type of sharing experienced by each page. This categorization is in turn based on an association between lock variables and the data they protect.

ADSM categorizes pages as falsely-shared, migratory, or producer/consumer(s). Migratory and producer/consumer(s) pages are managed in single-writer mode, while falsely-shared data are managed in multiple-writer mode. Multiple-writer pages are treated under the twinning and diffing mechanism [6], while single-writer pages do not require twins and diffs and are transferred as a whole. Coherence is kept with invalidations for most types of the shared data, but updates are used for lock-protected data in migratory state and barrier-protected data in producer/consumer(s) state. The first type of updates is sent solely to the processor acquiring the lock, while the second type of updates is sent to all the consumers of the data at a barrier. Note that both the categorization and adaptation techniques we propose are general and can be used in other software DSMs.

In order to evaluate our protocol, we performed experiments with 6 parallel applications on an 8-node SP2 system. We isolated the performance benefits of each of the main characteristics of our protocol and compared it against standard TreadMarks [16] and a version of TreadMarks that also adapts to sharing patterns [2]. The most fundamental difference between our adaptation strategy and the one in adaptive TreadMarks is that our algorithm requires no extra messages, while adaptive TreadMarks may require an excessive number of ownership transfer messages. In addition, ADSM uses update-based coherence for certain types of pages, while adaptive TreadMarks always invalidates pages.

Our results using the SP2 switch show that our protocol consistently outperforms its competitors. ADSM can improve the TreadMarks speedups by as much as 155%, while surpassing the performance of the adaptive TreadMarks implementation by as much as 67%. The reason for the superior performance of ADSM is that our protocol provides great reductions in communication traffic and memory and coherence overheads with respect to standard TreadMarks, while providing significantly better communication performance than adaptive TreadMarks.

Our main conclusions are that our categorization and adaptation strategies are useful techniques for improving the performance of page-based software DSMs, while ADSM is a highly-efficient software DSM system.

The remainder of this paper is organized as follows. Section 2 presents the main aspects of LRC, TreadMarks, and Adaptive TreadMarks, and serves as background material

for the rest of the paper. Section 3 describes the ADSM protocol. Given that the technique we introduce for categorizing the sharing patterns observed by pages can be applied by other DSMs, we dedicate it a full section of the paper; Section 4 describes the technique in detail. Section 5 presents our methodology and application workload. Experimental results are presented in section 6. Section 7 overviews the related work. Finally, section 8 summarizes our findings and concludes the paper.

2 Background

2.1 LRC and TreadMarks

The LRC algorithm [15] divides the program execution into intervals and computes a vector timestamp for each interval. This vector describes a happens-before-1 partial order between intervals of different processors [1]. On an acquire operation, the last releaser can determine the set of write notices (descriptions of the modifications made to shared data) that the acquiring processor needs to receive, i.e. the set of notices that precede the current acquire operation in the partial order. Upon receiving the notices, the acquirer can then change the state of its memory accordingly.

TreadMarks is an LRC protocol in which a write notice received represents an invalidation to the corresponding page. The propagation of the actual modifications made to a shared page is deferred until the acquirer suffers an access miss on the page. These modifications are determined using the twinning and diffing mechanism as follows. A page is initially write-protected, so that at the first write to it a violation occurs. On such a violation, an exact copy of the page (a twin) is made and the original copy of the page is made writable. When the actual modifications are required, the twin and the current version of the page are compared to create a runlength encoding of the modifications (a diff).

TreadMarks's use of twins and diffs allows processors to modify their local copies of a shared page simultaneously, thereby alleviating the negative impact of false sharing. More details about TreadMarks can be found in [16].

2.2 Adaptive TreadMarks

Amza *et al.* have proposed a version of TreadMarks that dynamically adapts the handling of shared pages between single-writer and multiple-writer modes. The multiple-writer part of the protocol uses the TreadMarks twinning and diffing mechanism, while the single-writer part uses an extension of the CVM single-writer protocol [14].

The single-writer mode uses the concept of page ownership and version numbers. The protocol only allows one writer (the owner) for a page at each point in time. Each

page has a version number which is incremented every time ownership is acquired. At the time of an acquire request, the owner replies with owner write notices for the pages it modified. Each owner write notice contains the processor id and the page's version number. On a write fault, the faulting processor requests the ownership of the page (and possibly the actual contents of the page) from the current owner. Note that this ownership transaction is necessary even when the page is already valid locally. On a read fault, ownership is not requested, the faulting processor simply asks for a copy of the page from the processor named in the owner write notice with largest version number. This processor might not be the current owner, but this is ok according to the definition of LRC.

Adaptation between the single and multiple-writer modes takes place dynamically as a processor handling a page in single-writer fashion checks for the occurrence of write-write false sharing of the page and switches to multiple-writer mode if this type of sharing is detected. Conversely, a processor switches from multiple-writer to single-writer mode if it detects the absence of write-write false sharing of the page.

The basic principle behind detecting write-write false sharing in single-writer mode is: there is no write-write false sharing of a page if and only if the processor taking a write fault and trying to get ownership knows the owner and the version number of the page. The principle behind detecting the absence of write-write false sharing in multiple-writer mode is: there is no write-write false sharing of a page if there is a write notice for the page that dominates all other write notices. More details on the adaptive implementation of TreadMarks can be found in [2].

3 ADSM

3.1 Overview

ADSM was built out of TreadMarks and improves on its performance by adapting to the applications' sharing patterns. The protocol implements two types of adaptation: a) it adapts between invalidate and update-based coherence maintenance strategies and b) it adapts between single and multiple-writer modes of operation.

These two types of adaptation are based on our dynamic categorization (called Sharing Pattern Categorization or simply SPC) of the type of sharing experienced by each page, which is described in detail in the next section. SPC is based on an association between each lock variable and the pages that experience faults inside of the critical sections the variable delimits. On a write fault inside a critical section, ADSM creates a write notice recording the page number and the id of the lock being held by the processor. In case of nested locks, the write notice is associated with

the lock most recently acquired by the processor. A write fault occurring outside of critical sections is represented by a write notice without a lock id. All write notices are associated with intervals, which are started at all synchronization operations. As in TreadMarks [16], processors in ADSM maintain vector timestamps that, on an acquire operation, allow the releasing processor to determine the set of write notices that the acquirer needs to receive.

SPC classifies pages as falsely-shared, migratory, or producer/consumer(s). Migratory and producer/consumer(s) pages are managed in single-writer mode, while falsely-shared data are managed in multiple-writer mode. Update-based coherence is used for lock-protected data in migratory state and barrier-protected data in producer/consumer(s) state. The transfer of updates of lock-protected data occurs on lock acquire operations in the following way: out of the migratory pages associated with the corresponding lock, the releaser initially determines the pages that have been modified since the acquirer last held the lock; these are the pages that will be sent to the acquirer as updates. The releaser then sends a lock grant message to the acquirer, including the write notices belonging to intervals not seen by the acquirer and the numbers of the pages to be updated. The actual updates follow the lock grant message in separate messages. Selective updates of lock-protected data have the potential to reduce the duration of critical sections without increasing the amount of data traffic.

Updates of producer/consumer(s) data are sent to all the consumers of the data at a barrier. The transfer of these updates is overlapped with the synchronization overhead of barriers in the following way: as soon as a processor determines that it is the producer of a producer/consumer(s) page, it starts storing the ids of the processors that request the page. When the processor gets to a barrier point, it sends a barrier arrival message to the manager, proceeds to determine the processors that should receive updates of its producer/consumer(s) pages, and, after this process is done, sends the actual updates. Even though processors do not have to wait for the updates to be received, at the barrier release point several producer/consumer(s) pages will already be updated at their consumers. Selective updates of barrier-protected data have the potential to reduce data access overheads without increasing the amount of traffic.

Just as in the adaptive version of TreadMarks, in ADSM the multiple-writer pages are treated under the twinning and diffing mechanism, while the coherence of single-writer pages is maintained by transferring whole pages. However, in contrast with adaptive TreadMarks, ADSM adapts between single-writer and multiple-writer modes of operation without the need for ownership messages. Furthermore, the categorization of sharing behavior in ADSM is detailed enough that can be used in adapting between invalidate and update-based coherence.

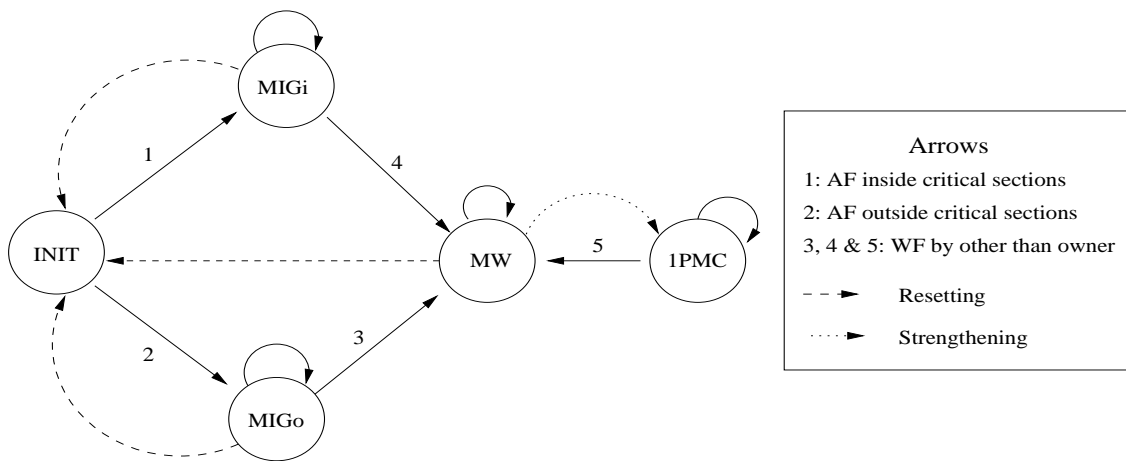


Figure 1. Page State Diagram for SPC. AF = Access Fault, WF = Write Fault.

3.2 Access Faults

Except where noted above, ADSM performs the same actions as TreadMarks on lock and barrier operations. Access faults however are treated somewhat differently in the two protocols.

The handling of read faults is dependent on the sharing pattern recently experienced by the page. On a read fault to a single-writer page, the offending processor must retrieve a new copy of the page from the processor that created the write notice most recently received for that page. As in TreadMarks, a read fault on a multiple-writer page causes the faulting processor to collect one or more diffs to bring the page up-to-date.

The handling of write faults is also dependent on the sharing pattern. A write fault on an access to a multiple-writer page is treated just like in TreadMarks, except for the fact that, if the fault happens inside a critical section, the write notice generated stores the id of a lock. Write faults to single-writer pages are treated differently. If the writer is the owner of the page, i.e. the processor currently allowed to be the single writer of the page, then no twinning is necessary; the processor simply creates a special type of write notice (referred to as “version notice” when necessary to differentiate between the two types of notices) that never has a diff associated with it. If the writer processor is not the owner of the page, it must change the state of the page according to SPC. Again, if the fault occurs inside a critical section, the respective lock variable must be identified in the write notice.

4 Sharing Pattern Categorization

The SPC strategy is a general technique for detecting the state of a page in terms of its sharing pattern. As aforemen-

tioned, the technique makes use of the association between lock variables and the data they protect and does not require any extra messages.

4.1 Basic States and Page Ownership

As shown in figure 1, a page in SPC can be in one of four main states: Initial (INIT), Migratory (MIG), Multiple Writer (MW), and One Producer/Multiple Consumers (1PMC), which also includes pages with one producer and one consumer. The MIG state is further subdivided into MIGi and MIGo for pages that experience faults inside and outside critical sections, respectively. Each processor maintains its own view of the state of the pages it caches. Changes in the state of a local page are determined by the events the processor observes about the page.

A MIGi page is faulted on inside critical sections protected by a single lock variable, a MIGo page is faulted on outside critical sections and by a single processor at a time (i.e. there is no false sharing of it), and a 1PMC page is always modified by the same processor. MW pages are the pages that do not fit any of these characteristics. MIG and 1PMC pages are considered single-writer pages. A single-writer page can only be modified by its current owner, after it retrieves the latest version of the page. Multiple-writer pages have no owner.

While 1PMC pages have fixed owners, the ownership of MIG pages should be transferred as appropriate. SPC implements these transfers without the need for any extra protocol messages. The ownership of a MIGi page is set to the lock that protects the critical section where the access fault occurred. Thus, any processor acquiring this lock will automatically get the ownership of the pages associated with it. When releasing the lock, the processor also releases the ownership of its pages.

The ownership of a MIGo page is transferred along with a copy of the page, if the processor servicing the page is its owner. If this is not the case, the requester only receives a copy of the page; the owner of the page remains the same as before. Although simple, our strategies for transferring ownership have proven very effective in practice.

4.2 State Transitions

Initially, all pages are in the INIT state, valid and owned by processor 0. On an access miss to a page in the INIT state, the system changes the page's state to MIGi (arrow 1 in figure 1) or MIGo (arrow 2) depending on where the processor was computing when the fault occurred. The ownership of the page is then given to the faulting processor.

A write performed to a single-writer page by its owner does not alter the state of the page. In case the writing processor is not the current owner of the page, a write fault to a single-writer page causes a state change to MW (arrows 3, 4, and 5) and prompts the creation of a twin for the page.

In order to adapt to sharing pattern changes, SPC may sometimes “strengthen” or “reset” the state of a page. Re-setting allows SPC to re-classify pages from scratch (INIT state). A MIG page is reset to INIT state in two cases: a) on the first access miss after a barrier by a processor other than the owner, if the page is invalid; or b) on the first page request received by the owner after a barrier operation, if the last version created before the barrier has not been sent to any other processor. The owner of the page remains the same, even when the state is in fact changed to INIT.

The state of a MW page can be strengthened to IPMC or reset back to INIT. On the first miss or page request to a MW page after a barrier, the offending processor determines, by inspecting the write notices, the access pattern to the page during the last two phases of execution (as delimited by the three preceding barrier events). If, during this period, the page has been modified by only one processor, its state is changed to IPMC. If it has been modified under a single lock, its state is changed to INIT. In both cases, the owner is set to the processor that created the last write notice. When changing the state of a page from MW to INIT or IPMC, a processor frees all the page's diffs and twins, thus reducing the memory requirements of the protocol.

The state of a page may be different in two distinct processors until they communicate. If they communicate via a transfer of write notices, the specific type of write notice received is relevant. Incorporating a write notice of a page locally in single-writer mode changes its local state to MW. Incorporating a version notice of a single-writer page does not change its local state. Incorporating a version notice changes a page's local state from MW to INIT, only if no write notice has been received for the page since the previ-

ous barrier, which means that the page has just been reset or strengthened.

If processors communicate via a page request the possible state transitions are different. The page's state is changed to MW at the processor servicing the request, if a) the state informed in the request message is MW, b) the page is invalid at the server (meaning that write-write false sharing of the page is taking place), c) the requester is trying to write a page that is locally considered IPMC at the moment, d) the requester is trying to write a page that is locally considered MIGo but the local processor is not the page's owner, e) the requester experienced the fault inside a critical section, but the page is locally considered MIGo, or f) the page is locally considered MIGi, but the requester experienced the fault inside a critical section protected by a different lock. Besides the actual contents of the page, the reply to the requester includes the new state of the page and its owner, as well as the associated lock for a MIGi page. At the end of the page request-reply pair, both the requester and server have the same state and owner for the page.

The worst-case scenario in terms of differences between the processors' views of page states is when processor X has the page in MW mode (and therefore computes diffs for it) and processor Y has the page in MIG or IPMC mode (and therefore transfers the whole page). However, this does not imply any loss of coherence, as the diffs can be merged with the latest single-writer version of the page to bring it up-to-date, when necessary. For example, if processor X has already created a diff for the page and receives a version notice coming from processor Y, processor X must, at the subsequent read miss to the page, request the new version of the page and then re-apply the locally-generated diffs to this new version.

4.3 Discussion

An important feature of ADSM is that it does not write-protect dirty copies of shared pages cached locally when acquiring and releasing locks. This lack of protection improves performance, but may hide the fact that some pages are written inside and outside of critical sections, or inside of critical sections protected by different locks. When these characteristics of pages are indeed hidden, SPC classifies them as single-writer, while they would be classified as multiple-writer had these characteristics been exposed. This misclassification is not a significant problem however, since, if these pages are actually written by multiple processors, SPC will quickly observe this fact. Besides, we have observed that the number of pages that would be temporarily misclassified as a result of this optimization is limited for most applications.

Note that SPC produces only a good approximation of the real state of pages, since SPC might temporarily catego-

Appl	Problem Size	Synchro	Seq Time
3D-FFT	$64 \times 64 \times 16$, 100 iters	barriers	55.90 secs
IS	$B_{max} = 2^{15}$, $N = 2^{23}$, 10 iters	locks and barriers	23.30 secs
TSP	18 cities	locks	20.40 secs
Water	512 molecules, 10 iters	locks and barriers	85.40 secs
MigDepth	512×256	locks	38.00 secs
MigFreq	512×256	locks	42.70 secs

Table 1. Application Characteristics.

rize a single-writer page as multiple-writer incorrectly. For instance, 1PMC pages are usually misclassified initially as MW pages, until page strengthening corrects the mistake. This effect is a result of our simple ownership transfer strategy and clearly has more benefits (in terms of avoiding extra ownership messages) than drawbacks, as our experiments demonstrate.

5 Methodology and Workload

Our experimental environment consists of an IBM SP2 with eight 66MHz Power2 processors. The nodes are connected by a 40 MBytes/s Omega switch. We experimented with ADSM, TreadMarks, and an adaptive version of TreadMarks. All systems communicate using the UDP protocol.

We report results for six parallel scientific applications. Four of them are from the TreadMarks distribution: 3D-FFT, Integer Sort (IS), TSP, and Water. These applications have been used in several previous evaluations of software DSMs, e.g. [2, 14]. The other two applications are seismic migration programs [12], MigFreq and MigDepth, from the Brazilian oil company, Petrobras. In table 1 we show the applications' problem sizes, synchronization styles, and sequential execution times. Applications were compiled with the -O2 option of the gcc and xlFortran compilers.

3D-FFT, originally from the NAS suite [10], solves a partial differential equation using forward and inverse FFT's. IS, also originally from the NAS suite, ranks an array of N integers using keys in the range $[0, B_{max}]$. TSP solves the traveling salesperson problem with a Branch-and-Bound algorithm. Water, originally from the SPLASH suite [20], simulates the dynamics of water molecules.

MigFreq performs 2D post-stack seismic migration using the $\omega - x$ algorithm. The program generates a 2D seismic section from a 2D array of seismic input data. Parallelization is done by assigning a block of frequencies to each processor. MigDepth solves the same problem as

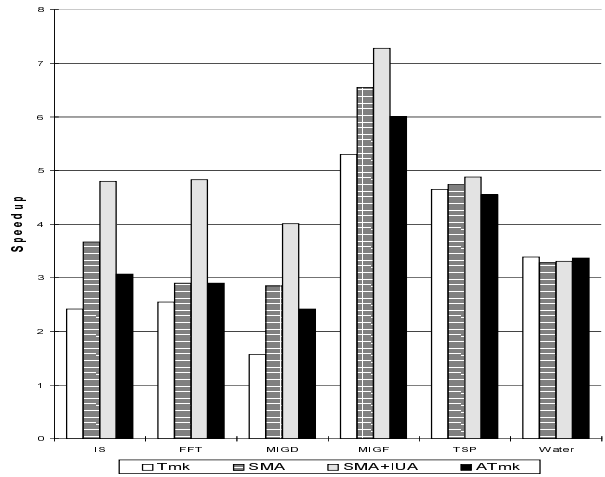


Figure 2. Application Speedups.

MigFreq, but uses depth partitioning, not frequency partitioning, to create parallel work. In this approach, each processor extrapolates the seismic section at a particular set of depths and the computation flows like a pipeline. This version achieves a higher degree of parallelism at the expense of much higher communication than in MigFreq.

6 Experimental Results

In this section we evaluate the performance benefits of each of the main characteristics of ADSM, while comparing the protocol against standard and adaptive TreadMarks implementations. We start with speedup results and then move on to a more detailed analysis of the performance of the protocol according to several metrics.

6.1 Speedup Performance

Figure 2 shows the speedup of our applications on an 8-node SP2 parallel machine communicating via its high-performance network. For each application we show, from left to right, the standard TreadMarks (“Tmk”), ADSM with single/multiple writer adaptation but without invalidate/update adaptation (“SMA”), ADSM (“SMA+IUA”), and adaptive TreadMarks (“ATmk”) performances.

Figure 2 demonstrates that two of our applications (MigFreq and TSP) exhibit good speedups under Tmk, while the other applications do not perform as well. SMA improves on the performance of Tmk significantly for three applications: IS by 52%, MigDepth by 82%, and MigFreq by 24%. The improvement for FFT is not as significant (14%), but is still non-trivial. The speedup of the other two applications is not affected by our adaptation between single and multiple-writer modes, even though this technique

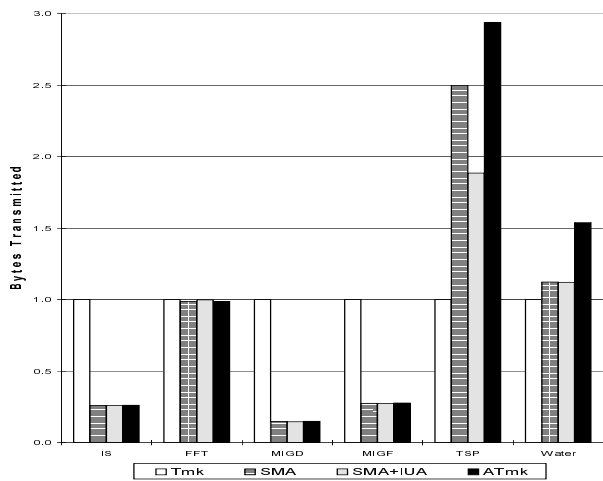


Figure 3. Number of Bytes Transferred.

does reduce the amount of memory and coherence (diffing and twinning) overheads in comparison to Tmk.

Throwing IUA into the mix, we get the full-blown ADSM (SMA+IUA) protocol. All but one (Water) of our applications take advantage of update coherence for certain data structures; IS, MigDepth, MigFreq, and TSP apply update coherence to lock-protected data in migratory state, while FFT applies updates to barrier-protected data in IPMC state.

The performance of ADSM shows that the addition of IUA also improves speedups significantly in most cases. The largest speedup improvements of ADSM with respect to SMA are for IS (31%), FFT (67%), and MigDepth (41%). More importantly however, we find that ADSM outperforms Tmk for all applications except Water; the speedup difference between ADSM and Tmk two protocols ranges from 5% for TSP to 155% in the case of MigDepth.

The comparison between ADSM and adaptive TreadMarks is also favorable to our protocol; ADSM outperforms adaptive TreadMarks for five of our applications by at least 7%. The advantage of our protocol is most significant for IS (56%), FFT (67%), MigDepth (66%), and MigFreq (21%). The speedup improvement for TSP is only 7%, while for Water Tmk, ADSM, and ATmk achieve roughly the same speedup.

The next few sections explain the speedups above.

6.2 Communication Traffic

Figures 3 and 4 present the number of Kbytes and messages, respectively, involved in each of our applications under the protocol versions we study. The order of the bars in the figures is the same as in figure 2. Each bar is normalized to the Tmk results.

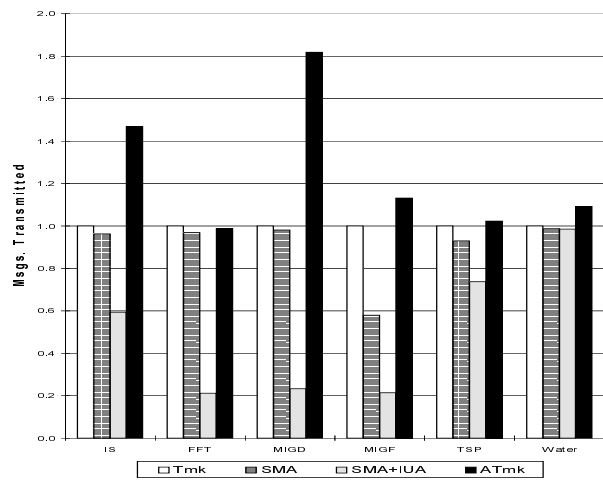


Figure 4. Number of Messages Transferred.

Figure 3 shows that SMA achieves enormous reductions with respect to Tmk in terms of the number of bytes transferred for IS (74%), MigDepth (85%), and MigFreq (73%). The reason for this result is that in these three applications most pages are completely re-written by a single processor every time they are touched. Computing diffs for these pages wastes time, space, and communication bandwidth, since diffs effectively overlap. Given that these pages are handled in single-writer mode under ADSM, these problems are eliminated.

For TSP, SMA leads to a significant increase in the number of bytes transferred during execution, since the amount of data modified in each page (diff) is small compared to the size of pages. Thus, for TSP, managing pages in multiple-writer mode saves on bytes transferred, even though most of the pages are written by one processor at a time. Water exhibits a similar behavior, but the increase in the number of bytes transferred is not as significant.

Full-blown ADSM transfers about the same amount of data as SMA in all cases except TSP, showing that our selective update technique is very effective and does not waste bandwidth in most cases. For TSP, ADSM actually reduces the amount of data transferred across the machine as a side effect of IUA's write-protection of MIGi pages sent as updates. These write-protections uncover the fact that several pages are written inside and outside of critical sections in TSP, which prompts SPC to classify these pages as MW.

ATmk transfers roughly the same amount of data as SMA and ADSM for four applications: IS, FFT, MigDepth, and MigFreq. For TSP and Water, ATmk transfers 56% and 37% more data than ADSM, respectively. The main difference between the protocols for these two applications is that ATmk correctly categorizes certain pages as single-writer, while ADSM misclassifies them as multiple-writer. This

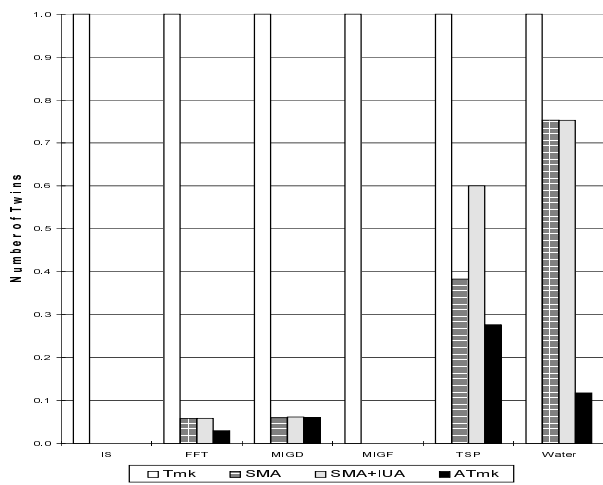


Figure 5. Number of Twins Generated.

misclassification turns out to be efficient in terms of communication, since the size of diffs is somewhat smaller than the size of pages, as mentioned above.

Figure 4 shows that Tmk and SMA transfer about the same number of messages for all applications, except MigFreq and TSP. For these applications, SMA provides 42% and 7% reductions, respectively, in the number of messages transferred. The reason for this result is that these applications exhibit a large number of cold access faults where both the page and diffs are required to bring the page up-to-date. Tmk requires at least two pairs of messages for this type of access fault, while under SMA only one request/reply pair is required, since these are single-writer pages in the application. The other applications also exhibit this type of behavior, but the number of these faults is not as significant.

Adding IUA to make ADSM produces large reductions in number of messages with respect to Tmk for five of our applications; reductions range from 26% for TSP to 79% for FFT and MigFreq. Again, this result confirms how extremely effective our update technique is at cutting down on communication traffic.

ATmk transfers between 53% and 95% more messages than SMA for three of our applications: IS, MigDepth, and MigFreq. For TSP and Water this difference is around 10%, while for FFT SMA and ATmk transfer roughly the same number of messages. The extra message transfers in ATmk are almost entirely due to ownership-related (request and reply) messages. For MigDepth, for instance, SMA transfers 76697 messages, while ATmk transfers 142205 messages out of which 65224 are ownership-related messages. As another example, take IS. For this application SMA transfers 10192 messages, while ATmk transfers 15545 messages out of which 5346 are ownership-related messages.

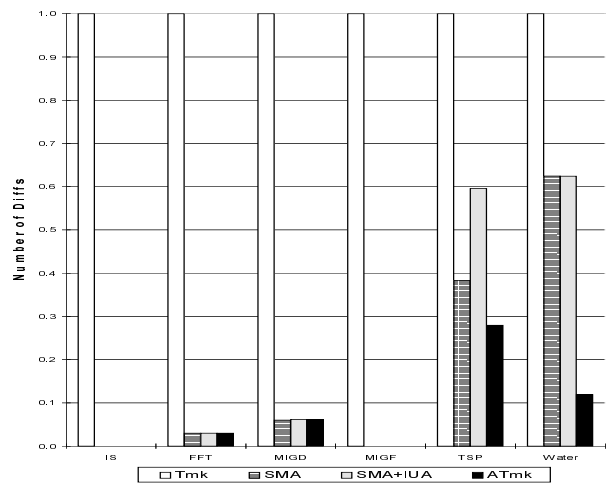


Figure 6. Number of Diffs Generated.

In comparison to ADSM, ATmk always transfers more messages; the difference ranges from 10% (Water) to a factor of 8 (MigDepth). This result is due to the fact that in ATmk each access fault requires a request/reply pair of messages or possibly a pair of ownership-related messages, while in ADSM several access faults are prevented via updates (as shown in section 6.5) and no ownership messages are necessary. Note that each of our update messages can carry up to 15 pages and does not require a reply.

6.3 Coherence Overhead

Figures 5 and 6 present the number of twins and diffs, respectively, involved in each of our applications under the protocol versions we study. The order of the bars in the figures is again the same as in figure 2. Each bar is normalized to the Tmk results.

The figures show that SMA provides enormous reductions in coherence overhead with respect to Tmk for our applications, as they all exhibit a significant fraction of single-writer pages. Reductions are most significant for IS and MigFreq, for which SMA is able to eliminate all twins and diffs. For the other applications, reductions in the number of twins range from 25% (Water) to 94% (FFT and MigDepth), while reductions in the number of diffs range from 38% (Water) to 97% (FFT). ADSM generates the same number of twins and diffs as SMA in all cases, except TSP where the additional write-protections involved in IUA cause a larger number of pages to be managed under the twinning and diffing mechanism.

ATmk generates roughly the same amount of coherence overhead as ADSM for three of our applications (IS, MigDepth, and MigFreq), while generating fewer twins for FFT, and fewer twins and diffs for TSP and Water. For FFT, ATmk incurs many fewer twin operations than ADSM, even

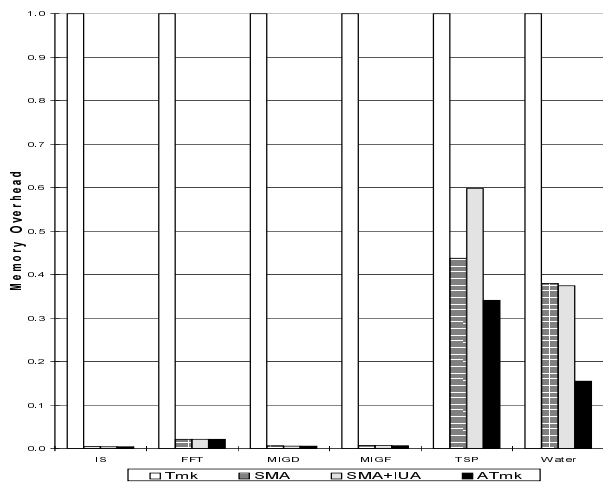


Figure 7. Memory Overhead.

though the number of diffs generated by the two protocols is about the same. The reason for this result is that SPC *initially* miscategorizes certain pages as MW when they are in fact 1PMC. This leads to more twins but not more diffs, since SPC corrects its mistake after two barrier events. For TSP and Water, ADSM also misclassifies single-writer pages as MW, since some of these pages are modified inside and outside of critical sections (TSP) or modified inside critical sections protected by different locks during a phase (Water).

6.4 Memory Overhead

Figure 7 shows the memory overhead for each of the protocol versions we study. The order of the bars in the figure is again the same as in figure 2. Each bar is normalized to the Tmk results.

We define the memory overhead to be the number of bytes used to store diffs, twins, intervals, and write notices. Note that in our experiments we turned all garbage collection off. Garbage collection would reduce the memory overhead of Tmk, but would also require extra messages to reallocate. We resolved this tradeoff in favor of fewer messages. For the other protocol versions we study, garbage collection is not as necessary, since they reduce the memory consumption to acceptable levels. The largest consumption involved in ADSM is for Water, where the memory overhead per processor is less than 1.3 MBytes.

The figure shows that SMA improves the Tmk memory consumption, with significant memory overhead reductions coming from not requiring twins and diffs for single-writer pages. The technique provides improvements in overhead with respect to Tmk for all applications, ranging from 56% (TSP) to 99% (IS, MigDepth, and MigFreq). The memory overhead involved in full-blown ADSM is about the same as

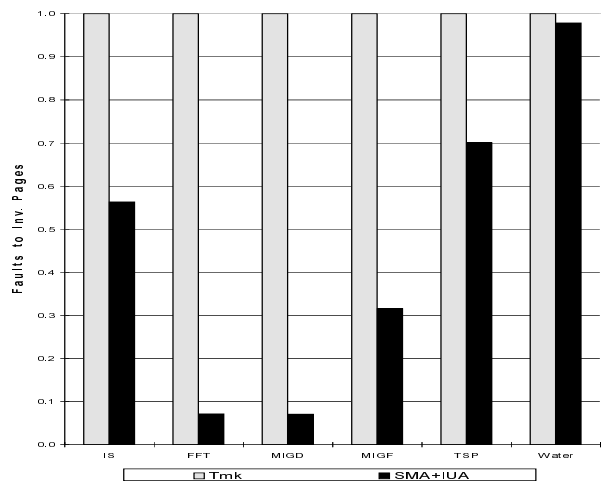


Figure 8. Num Accesses to Invalid Pages.

in SMA, except for TSP where IUA entails a larger number of MW pages.

ATmk entails less overhead than ADSM for two applications: TSP and Water. The explanation for this result is the more accurate page sharing characterization of ATmk for these applications.

6.5 Data Access Overhead

Figure 8 presents the number of access faults to invalid pages (i.e. the faults that require external communication in Tmk) incurred by each application under two of the protocols we study, Tmk and ADSM. We do not include results for the other protocol implementations since they have no effect on the number of access faults incurred by Tmk.

The results shown in this figure demonstrate that the IUA technique reduces the number of faults tremendously with respect to the other protocol implementations, except for Water where most pages are classified as either MW or MIGo. Reductions range from 30% for TSP to 93% for FFT and MigDepth. The reduction in the number of misses is most significant for FFT and MigDepth, but for different reasons. In FFT, the reduction comes from the fact that most pages undergo 1PMC sharing during the matrix transpose phase of the algorithm, while in MigDepth all pages are accessed inside of critical sections, being classified as MIGi.

6.6 Discussion

Single-writer pages can frequently be found in parallel applications. The results just presented show that four of our applications (IS, FFT, MigDepth, and MigFreq) are dominated by single-writer pages, while the other two (TSP

and Water) exhibit a mixture of single and multiple-writer pages. ADSM's efficient treatment of the different types of single-writer pages allows it to achieve better performance than Tmk and ATmk. For applications dominated by multiple-writer pages, Tmk, ATmk, and ADSM perform about the same, since the latter two protocols add very little overhead to the treatment of this type of pages.

SPC produces an approximation of the sharing behavior of applications. A comparison with the precise categorization of ATmk shows that, for TSP and Water, SPC misclassifies certain single-writer pages as MW. These misclassifications occur for pages that are effectively written by a single processor at a time, but are modified inside and outside of critical sections (TSP) and modified inside critical sections protected by different locks during a phase (Water).

Note however that the tradeoff between SMA and ATmk is directly affected by the elimination of ownership messages in SMA, which in certain cases leads to significant performance improvements. A comparison against the single-writer-based version of CVM [14] would also have been favorable to ADSM, since, CVM also involves extraneous ownership-related messages.

In essence, the tradeoff between ADSM and ATmk is about the lower communication and access fault overheads of ADSM against the lower coherence and memory overheads of ATmk. Given that in software DSMs optimizing communication and reducing access faults is the most effective way of improving performance, ADSM compares favorably against ATmk.

7 Related Work

The work by Amza *et al.* [2] on adaptive TreadMarks implementations is the most similar to ours; they use the same base system and also adapt between single and multiple-writer modes. In this paper, we compare their most important implementation¹ against ADSM.

Several systems, besides ADSM, combine or adapt between multiple coherence protocols, e.g. [6, 8, 9, 13, 17, 19, 21]. Munin [6] was the first system to propose the use of multiple protocols for different shared data objects. The choice of protocol is left to the user, however. ADSM avoids burdening the programmer by adapting automatically.

Just as ADSM, the Lazy Hybrid (LH) protocol studied by Dwarkadas *et al.* in [9] applies a hybrid invalidate/update coherence approach. During a lock acquire operation in the LH protocol, the releaser determines the diffs it has that correspond to modifications not yet seen by the acquirer and sends them along with the lock grant message. In addition,

¹[2] also includes a version of TreadMarks that adapts itself according to write granularity. This version is shown to lead to minor improvements with respect to the version that only adapts according to write-write false sharing.

just before barrier arrivals, each processor sends diffs to all processors that cache pages modified locally. ADSM differs from LH mainly in that updates are only used for specific types of data: MIGi pages associated with the lock variable on a lock acquire and the 1PMC pages on a barrier operation. The performance benefits of the updates in the LH protocol were found to be limited, while our update strategy is clearly beneficial in most cases.

In a recent paper [17], Kim and Vaidya proposed a software DSM where nodes periodically choose between invalidate, competitive, or migratory protocols on a per page basis, depending on the estimated cost of using each of the protocols. In their system a page managed in migratory fashion is not allowed to have multiple concurrent readers, while in ADSM these pages can and often do have multiple concurrent readers. Further comparisons between ADSM and their system are difficult to make however, since neither the overall performance of the DSM nor the overhead of cost estimation is evaluated in [17].

ADSM uses an approximate association between locks and the data they protect. Systems based on Scope [13] and Entry Consistency [4], such as Brazos [21] and the Affinity Entry Consistency (AEC) protocol [19] also associate locks and data, but they must do it for correctness, while ADSM only uses its lock/data association to adapt to sharing patterns. The Brazos system and the AEC protocol also use a combination of invalidates and updates to keep shared data coherent.

Prefetching techniques attempt to achieve the same benefits as update-based approaches. A few studies [3, 5, 8, 18] have evaluated prefetching techniques for software DSMs. In the Dynamic Aggregation (DA) technique [3], for instance, nodes compute page groups (sets of pages that are accessed in between synchronization points) at each synchronization. The diffs for all pages of a group are requested on the first fault on any of the group's pages. The extent to which prefetching techniques can outperform update strategies such as IUA is one of the subjects of our future work.

Several researchers have proposed techniques for adapting to sharing patterns in the context of hardware DSMs, e.g. [7, 11, 22, 23]. Cox and Fowler [7] and Stenström *et al.* [22] have looked at optimizing the handling of migratory data in the coherence protocol. Dahlgren and Stenström [11] have studied a hybrid invalidate/update protocol where each processor makes a local decision to invalidate or update a cache block when it receives an update message. Trancoso and Torrellas [23] have proposed a system where a processor releasing a lock forwards the data it modified inside the critical section to the next acquirer of the lock. In contrast, ADSM implements more sophisticated sharing categorization and adaptation techniques than would be possible/efficient for hardware DSMs.

8 Summary and Conclusions

In this paper we proposed the Adaptive DSM (ADSM) protocol, which relies heavily on the Sharing Pattern Categorization (SPC) technique for adapting between single and multiple-writer handling of shared pages and between invalidate and update-based coherence.

Our analysis of ADSM showed that the protocol is very successful at reducing the communication, coherence, and memory overheads of most applications with respect to standard TreadMarks. Our comparison against an efficient adaptive TreadMarks implementation was favorable to ADSM, as a result of its better communication performance and selective updates.

In summary, our main contributions have been the proposal and evaluation of SPC, our adaptation strategies, and ADSM. ADSM has been shown an efficient software DSM protocol, while SPC and our adaptation techniques have been shown effective techniques that can be used in several software DSMs.

Acknowledgements

The authors would like to thank the adaptive TreadMarks group, in particular Cristiana Amza, for letting us borrow the implementation studied in this paper. We would also like to thank Paulo Osório Figueiredo and Silvio Sinedino Pinheiro for their help with the seismic migration applications. Comments by Cristiana Seidel, Raquel Pinto, and Maria Clicia de Castro helped improve the presentation of the paper. We also acknowledge the support of the staff of the Brazilian National Laboratory for Scientific Computing (LNCC).

References

- [1] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, June 1993.
- [2] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt Between Single Writer and Multiple Writer. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, February 1997.
- [3] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.
- [4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the '93 Comp-Con Conference*, February 1993.
- [5] R. Bianchini, R. Pinto, and C. L. Amorim. Page Fault Behavior and Prefetching in Software DSMs. 1997. Submitted for publication.
- [6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [7] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.
- [8] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [9] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.
- [10] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
- [11] F. Dahlgren and P. Stenström. Reducing the Write Traffic for a Hybrid Cache Protocol. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.
- [12] P. Figueiredo. Exploitation of Parallelism in Seismic Migration. Master's thesis, University of Illinois at Urbana-Champaign, April 1995.
- [13] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.
- [14] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.
- [15] 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*, May 1992.
- [16] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, January 1994.
- [17] J.-H. Kim and N. H. Vaidya. Adaptive Migratory Scheme for Distributed Shared Memory. In *Proceedings of the 1997 International Conference on Supercomputing*, August 1997.
- [18] M. Karlsson and Per Stenström. Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems. *Journal of Parallel and Distributed Computing*, September 1997.
- [19] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing*, August 1997.
- [20] 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.
- [21] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1997 USENIX Windows/NT Workshop*, August 1997.
- [22] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.
- [23] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing*, August 1996.