On the Doubly-Linked List Protocol for Distributed Shared Memory Multiprocessor Systems

Albert C.K. Lau, Kelvin H.W. Leung, Nelson H.C. Yung and Y.S. Cheung

Department of Electrical and Electronics Engineering
University of Hong Kong
Haking Wong Building, Pokfulam Road, Hong Kong
email: acklau@hkueee.hku.hk, nyung@hkueee.hku.hk

Abstract

This paper introduces the Doubly-Linked List (DLL) Protocol for Distributed Shared Memory (DSM) Multiprocessor Systems. The protocol makes uses of two linked list to keep track of valid copies of pages in the system, thus eliminating the use of copy-sets. Simulation studies show that the DLL protocol achieved considerable speed-up for common mathematical problems including a linear equations solver and a matrix multiplier. Performance improvement of up to 51.9% over the Dynamic Distributed Manager algorithm is obtained. Further improvement and possible modification of the protocol will also discussed.

1. Introduction

Distributed Shared-Memory (DSM) [1] is becoming an important aspect of Massively Parallel Processing (MPP) because it allows programmers to use the shared-memory programming model, which is much more manageable than the message-passing model used by traditional Massively Parallel Processors. The Doubly-Linked List (DLL) protocol discussed in this paper is a software DSM algorithm that is suitable for implementation in distributed operating systems of modern MPPs. A typical MPP configuration consists of a large number of processing nodes connected together by an interconnection network. The protocol presented in this paper, however, works for a more generalized hierarchical cluster model [2], which consists of multiple clusters connected by an interconnection network (*Figure 1a*). Each of these clusters may then have a small number of Processing Elements (PEs), its own local memory, and perhaps a dedicated Communication Processor (CP) (*Figure 1b*). The typical MPP configuration may be considered as a special case of this model in which the number of PEs in a cluster equals to 1. DSM is particularly important in this kind of architecture

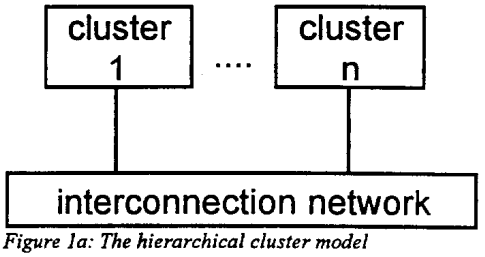

Figure 1a: The hierarchical cluster model

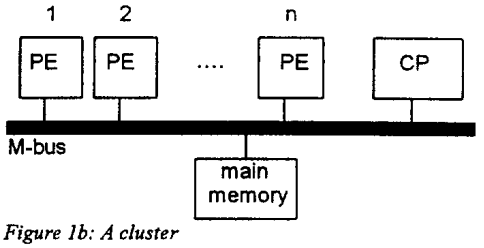

Figure 1b: A cluster

because it is difficult for programmers to handle both the shared-memory and the distributed-memory model at the same time. It is very desirable to hide this complication from the programmers so that they only see a uniform shared-memory model. On the other hand, since memory within a cluster is physically shared by the PEs, the overhead of shared memory accesses within a cluster is minimal. In order to create a shared-memory environment out of the physically distributed memory, a protocol is needed to handle remote memory accesses as well as to maintain memory coherence.

Ivy [3], one of the first transparent DSM systems, implemented DSM as virtual memory. In this system, when a page fault occurs in a cluster's local memory, instead of loading from disk, the faulting page is fetched from a remote cluster that has a valid copy of the page. It experimented with various DSM algorithms and concluded that the Dynamic Distributed Manager (DDM) algorithm generally had

0-7803-2018-2/95/$4.00

the best performance. The Fixed Distributed Manager algorithm, also proposed in the same paper, was later used in the Intel iPSC/2 hypercube multicomputers [4]. In the DDM algorithm, pages are migrated freely throughout the system and replicated as needed for shared read accesses by different clusters. The page management is performed by individual owner cluster of a page that keeps the copy-set (the set of clusters that has valid copy of the page). Whenever there is a write access to a page, the owner of the page invalidates all other copies of the page in the system by making use of the copy-set, then transfers the ownership to the cluster that writes to the page.

The DDM algorithm has certain advantages. First, it is simple and easy to implement, thus can be added to existing systems with minimum effort. Second, since it is an extension of the basic virtual memory system, which is a standard feature supported by virtually all contemporary microprocessors, the overhead caused by the algorithm is small. Third, it is fully transparent to the programmer, so systems with different underlying architecture can use the same simple programming model.

However, there are areas in the DDM algorithm that can be improved so that it fits better in modern MPPs. In fact, the original DDM algorithm was implemented on a network of Apollo workstations. Therefore, the idea of storing the copy-set in the owner and having it invalidate other copies in the system worked satisfactorily. However, given the high speed interconnection network used by most modern MPPs today, the burst of invalidation messages generated by the owner can cause network congestion around the cluster and degrade the network performance significantly. Also, the size of the copy-set varies and can be as large as the number of nodes in the system in the worst case. This greatly limits the scalability of the algorithm because as the system grows larger, the amount of memory allocated for the copy-sets becomes impractically large. In Li's paper [3], a method to partially distribute the copy-set using trees of clusters were proposed. In this paper, the idea is further developed into the Doubly-Linked List (DLL) Protocol.

The concept of the DLL protocol is built on the existence of two types of links for each page, namely the N-links and the P-links. The N-links are used to locate the current owner of a page and is similar to the probable owner field in the DDM algorithm [3]. The P-links are used to maintain a linked list of clusters that contain valid copies of the page. In other words, following the P-links, we can locate all valid copies of the page in the system. The purpose of this approach is to fully distribute the copy-set using the P-links.

There are several advantages using a linked list to maintain the copy-set. First, it is simple and it reduces the number of messages needed for invalidation nearly by half. Second, only a small constant storage space is needed to store a link, as compared to the varying space needed to store the copy-set or the tree nodes in [3]. Third, since the copy-set is completely distributed, invalidation is not likely to cause congestion in the interconnection network.

The DLL protocol is explained in detail in the next section. In addition, a feature will be proposed to further enhance the performance of the basic DLL protocol. Furthermore, the performance of the DLL protocol will be compared to various other algorithms by extensive simulation studies. Finally, possible further research on the DLL protocol will be described.

2. The Basic Doubly-linked List Protocol

In the DLL protocol, each cluster has its own page table, which contains information about all memory pages in the system. Each memory page in the page table can have one of the three states -- E (exclusive), S (shared) or I (invalid). E state means the cluster has the only copy of the page in the whole system. S state means more than one cluster in the system have copies of the page. I state means the cluster does not have a valid copy of the page.

Every page has an owner, although page ownership is frequently transferred between clusters. The owner of a page is the cluster that most recently acquired the page. It is the responsibility of the owner to supply the page to requesting clusters.

Also contained in the page table are two links for each page -- the P-link and the N-link. The P-link points to the cluster that is the previous owner of the page, while the N-link points to the cluster to which the page ownership is given, i.e., the new owner of the page. A null N-link means the cluster is the owner of the page.

When the system is initialized, memory pages are distributed to each cluster's local memory in an interleaved fashion, i.e., pages p0, p4, p8, ... go to cluster c0, pages p1, p5, p9, ... go to cluster c1 and so on. The cluster that contains a page in its local memory at system startup is the initial owner of the page. The page table is initialized as follows: the state of a page is E in its owner's page table, and I in other clusters' page table. The P-links of all pages in every

cluster are set to null. The N-link of a page is null in its owner's page table, and points to the owner in other clusters' page table. For example, the initial state of part of the page table in c0 is as shown in

	State	P-link	N-link
p0	E	null	null
p1	I	null	c1
p2	I	null	c2

Table 1: Initial state of c0's page table

table 1.

Read and write accesses performed to pages with different states will initiate different course of events. They will be explain below.

2.1. Read accesses to E pages

If a page is in E state, the cluster has a valid copy of the page and read accesses to the page can be handled locally. No messages will be sent to other clusters and the page table will not be changed.

2.2. Read accesses to S pages

Read accesses to S pages are handled exactly the same way as read accesses to E pages.

2.3. Read accesses to I pages

In this case, the local memory of the cluster does not have a valid copy of the page being accessed so a copy of the page must be obtained from the owner. A `read-request` (RR) message will be sent to the cluster pointed to by the N-link in the page table, requesting for the missing page. If the cluster's local memory has no room for the new page, the replacement algorithm, which will be described later in this paper, will be used to make room for the new page.

If the cluster receiving the request is not the owner of the page, it will forward the RR message to the cluster pointed to by the N-link of the page in its own page table. This process repeats until the owner of the page is reached.

On receiving the RR message, the owner will send a `read-data` (RD) message containing the requested page back to the requesting cluster. Then, it will set the N-link of the page in its own page table to point to the requesting cluster, thus transferring the ownership of the page to the requesting cluster. Finally, it sets the state of the page to S.

The requesting cluster, on receiving the RD message, copies the page to its local memory. Then,

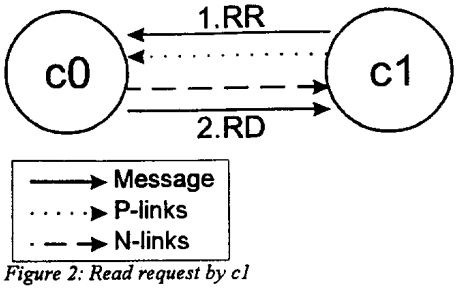

Figure 2: Read request by c1

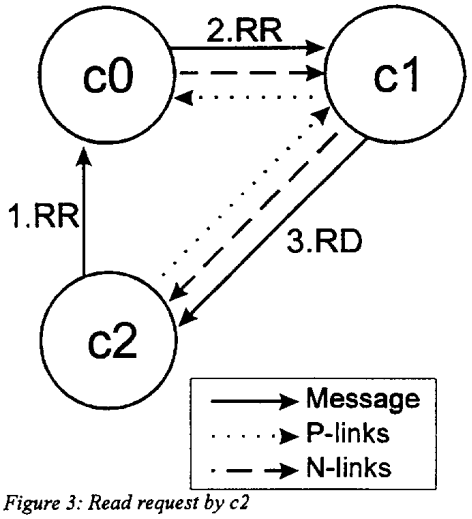

Figure 3: Read request by c2

it sets the P and N-links of the page to point to the replying cluster (i.e., the previous owner of the page) and to null, respectively. It becomes the new owner of the page and changes the page state from I to S.

The following are examples of read requests. In the examples, we shall assume a small system with 4 clusters, c0-c3. Initially, c0 is the owner of memory page p0. Both the P and N-links of p0 in c0 are set to null and its state set to E. In all other clusters, the P-link of p0 are null, N-link is c0 and the state is I.

Now, assume c1 tries to perform a read access to p0. An RR message is sent from c1 to c0. On receiving the message, c0 sends a RD message, containing p0, back to c1, then sets its own N-link to c1, and changes the page state to S. When c1 receives the RD message, it copies the page to its local memory, sets its P-link to c0 and N-link to null, and change the page state to S. The process is depicted in *Figure 2*. In all the figures, the solid arrows represent the messages passed between clusters, while the dotted and dashed arrows show the state of the P and N-links after the whole process has been completed.

In the event of another cluster c2 performs a read access to p0, as the state of p0 in c2 is I, c2 sends an RR message to the cluster pointed to by its N-link, i.e., cluster c0. When c0 receives the RR message, since it is no longer the owner of p0 (N-link not null), it forwards the message to the cluster pointed to by its own N-link, i.e., cluster c1. When c1, which is the current owner of p0, receives the RR message, it sends an RD message back to the requesting cluster c2, and then set its N-link to c2, thus transferring the ownership of p0 to c2. Cluster c2, on receiving the RD message from c0, copies p0 into its local memory, changes the state of p0 to S, and sets the P and N-links to c1 and null, respectively. The cluster c2 has become the new owner of p0. The process is depicted in *Figure 3*.

The current states of the p0 entries of the page tables in cluster c0, c1 and c2 are summarized in *Table 2*. At this point, c0, c1 and c2 each has a copy of p0 in state S. Therefore, read accesses performed by these clusters can be handled locally.

	State	P-link	N-link
c0	S	null	c1
c1	S	c0	c2
c2	S	c1	null

Table 2: State of p0 in c0, c1 & c2

2.4. Write accesses to E pages

Since the cluster contains the only copy of the page in the whole system, it can write to the page without generating any messages nor changes of the page table.

2.5. Write accesses to S pages

When more than one cluster contain copies of the page, write access in one copy causes the other copies to become obsolete. The DLL protocol uses a **write-invalidate** algorithm to solve this problem because it generates fewer traffics [1, 5]. The **write-update** algorithm used by some other systems [6] is not suitable for this implementation of the DLL protocol which used the sequential consistency model [2], owing to the high cost of the write-update messages. The possibility of using write-update in future implementation of the DLL protocol using relaxed consistency models will be discussed in Section 5 - Further Research on the DLL protocol.

When a cluster performs a write access to an S page, a `write-invalidate` (WI) message will be sent to the cluster pointed to by the N-link.

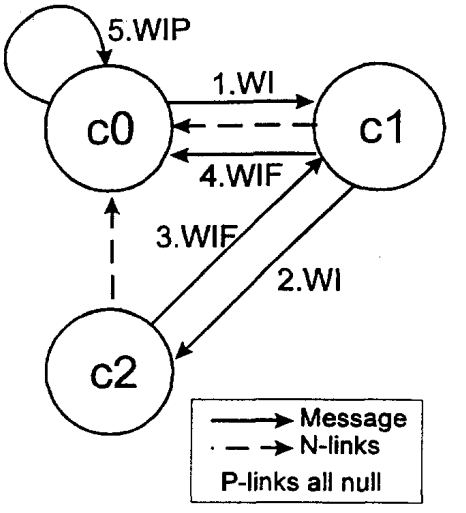

Figure 4: Write request by c0

Following the series of N-links, the message will eventually reach the owner of the page. The owner, on receiving the WI message, will send a `write-invalidate-forward` (WIF) message to the cluster pointed to by its P-link, change the page to I state, and reset its P and N-links to null and to the requesting cluster, respectively. All clusters receiving the WIF message will forward the message to the cluster pointed to by its own P-link, change the page to I state, and reset its P-link to null and N-link to the requesting cluster. The requesting cluster will also receive the WIF message. It will just ignore the message and forward it to the cluster pointed to by its P-link. Following the P-links, all copies of the page in the system, except the one in the requesting cluster, will be invalidated. When the WIF message reaches the cluster whose P-link is null, the cluster will send a `write-invalidate-performed` (WIP) message to the requesting cluster.

The requesting cluster, upon receiving the WIP message, will set its P and N-links of the page to null and change the state of the page to E. It becomes the new exclusive owner of the page. At this point, the write access can be performed.

An example of a write request to an S page is as follows. Assume the state of p0 in each cluster is as shown in *table 2* and now c0 performs a write access to p0. Since p0 is in state S in c0, other clusters that have copies of c0 must have their copies invalidated. Therefore, a WI message is sent to the cluster pointed to by the N-link, i.e., cluster c1. Following the N-links, c1 forwards the WI message to c2, which is the current owner of p0. Cluster c2 then sends a WIF message to the cluster pointed to by its

P-link, i.e., cluster c1, changes the state of its p0 to I, and resets its P-link to null and N-link to c0. The cluster c1, upon receiving the WIF message, changes the state of p0 to I, forwards the message to the cluster pointed to by its own P-link, i.e., cluster c0, and reset its P and N-links to null and c0, respectively. When c0 receives the WIF message, as it is the requesting cluster, it ignores the message. Since c0's P-link is null, all copies of p0 in the system, except the one in c0, are invalidated. At this point, c0 should send a WIP message back to the requesting cluster. In this case, however, the requesting cluster is c0 itself, so this message is skipped. Finally, c0 changes the state of p0 to E, sets both of its P and N-links to null, and completes the write access. The cluster c0 becomes the new exclusive owner of p0. The process is depicted in *Figure 4*.

2.6. Write accesses to I pages

Write access to an I page is handled in a way similar to handling write access to an E page, except in this case, the requesting cluster does not have a valid copy of the page. Therefore, the page must be copied from its current owner and if the requesting cluster has no space for the page, the replacement algorithm must be used to make room for it.

When a write access to an I page occurs, the cluster sends a `write-request` (WR) message to the cluster pointed to by its N-link. Following the N-links, the message will eventually reach the owner of the page. The owner, on receiving the WR message, will perform three actions. First, a `write-data` (WD) message, containing a copy of the page, will be sent to the requesting cluster. Second, a `write-invalidate-forward` (WIF) message will be sent to the cluster pointed to by its P-link. Third, it invalidates its own copy of the page and resets its P and N-links to null and to the requesting cluster, respectively.

Following the P-links, the WIF message will go through every cluster that has a copy of the page, which will also invalidate its own copy of the page and reset their P-link to null and N-link to the requesting cluster. Finally, when the WIF message reaches the cluster with a null P-link, that cluster will send a `write-invalidate-performed` (WIP) message back to the requesting cluster.

When the requesting cluster receives both the WD and the WIP message, it sets its P and N-links to null and change the state of the page to E. It becomes the new exclusive owner of the page and the write access can be performed.

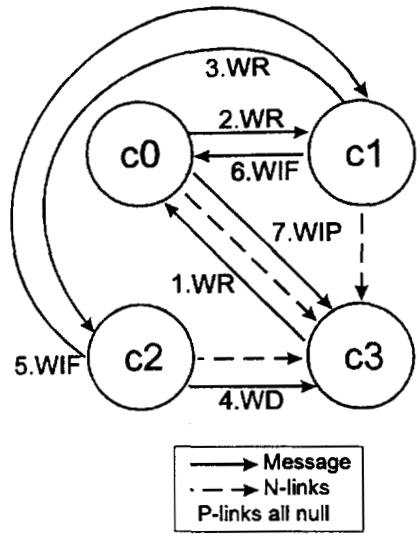

Figure 5: Write request by c3

Let us consider an example of a write request to an I page. Again assume the state of the system is as shown in *Table 2*. Now, another cluster, c3, tries to perform a write access to p0. Since the state of p0 in c3 is I, c3 sends a WR message via the N-links to the owner of p0, i.e., c2 (*Figure 5*).

When c2 receives the WR message, it first sends a WD message, which contains a copy of p0, to c3. Second, it sends a WIF message to the cluster pointed to by its P-link and changes the state of its own p0 to I. Third, it resets its P and N-links to null and to c3, respectively.

The WIF message, following the P-links, goes through every cluster that contains a copy of p0, i.e., c1 and c0, which also changes the state of p0 to I and reset the P and N-links to null and to c3, respectively. When the WIF message reaches c0, whose P-link is originally null, c0 will send a WIP message to c3.

When c3 receives both the WD and the WIP messages, it copies p0 into its local memory, and set both the P and N-links to null. The cluster c3 becomes the new exclusive owner of p0. The process is depicted in *Figure 5*.

2.7. Replacement Algorithm

When there is no room in a cluster's local memory for a newly requested page, one of the pages currently in the local memory must be replaced and swap out. To replace a page, two problems must be

considered [1]: Which page should be replaced? Where should the replaced page go?

The first problem is similar to the replacement problem in multi-processor caches. In this case, a prioritized LRU (least recently used) algorithm is used. Highest priority is given to those S pages whose owners are not the replacing cluster, as nothing needs to be done to invalidate these pages. Second priority is given to those S pages whose owner is the replacing cluster. Replacement of one of these pages involves the transfer of ownership of the page to the cluster pointed to by the P-link of the replacing cluster. The lowest priority is given to E pages because another cluster must be found to store the replaced page.

This leads to the second problem: Where should the replaced page go? One way is to keep track of free memory in system and swap out the page to a cluster with enough space. In the DLL protocol, however, since there is no centralized memory manager, it is very expensive to keep track of free memory. Moreover, the case that an E page must be replaced should be very rare, so we can afford to use a more costly method to find the new owner. Therefore, when an E page must be replaced, the replacing cluster will just send it to the next cluster, i.e., the nearest cluster. The cluster receiving the page becomes the new owner of the page, even if it might have to replace one of its own page to make room for the replaced page. This method is guaranteed to work given the virtual memory size is smaller than or equal to the physical memory size. Of course, if secondary memories such as disks are available for the storage of swap-out pages, we do not have this limitation.

3. Performance Enhancement Feature

In this section, an enhancement feature called **N-links reduction** will be described. The feature, when incorporated into the basic DLL protocol, will reduce the overhead caused by inter-cluster communication and thus enhance the overall performance.

3.1. N-links Reduction

In the basic protocol, when a cluster issues a read request, it sends the message to the cluster pointed to by its N-link. The message may then need to go through a number of unrelated clusters, which are the previous owners of the requested page, before it reaches the true owner. This may introduce a significant amount of overhead as the N-links grow longer.

A simple method to reduce this overhead is to periodically broadcast the page ownership to all clusters in the system, so that all read-request to the page can reach the owner directly without going through unrelated clusters. This method, however, introduces new problems of its own. First, it is difficult to determine how often the ownership should be broadcast. If a fixed time interval is used, a lot of unnecessary broadcast might be generated. One way is to count the number of unrelated clusters that a `read-request` message goes through before reaching the owner, and broadcast the ownership if the number is greater than a certain threshold. The second problem is that broadcasting itself generates a lot of traffic. As a page may not be used by all clusters in the system, many of these broadcasting may be unnecessary.

An alternative method is to reduce the N-links for every read request to a page. According to the DLL protocol, the cluster that generates the read request will become the new owner of the page after the request has been serviced. Therefore, all the clusters that are involved in forwarding the `read-request` message may change their N-links to the requesting cluster, even though the request has not yet been completed. The requesting cluster should lock the page and queue all accesses to it until the `read-data` message is received. Although this method only partially reduces the N-links (only the N-links of clusters that are previously involved in forwarding a message are reduced), it is virtually free because it only uses the normal `read-request` message without adding new information to it. The performance of the protocol with N-links Reduction will be compared to the original protocol using simulation studies.

4. Simulated Performance of the DLL Protocol

For the purpose of simulation studies, we have implemented three different DSM algorithms. Apart from the DLL protocol, a version of the Central Server algorithm and the DDM algorithm are also implemented. In the Central Server algorithm, all the page information and remote memory accesses are handled by a central server, which is one of the cluster in the system. In the DDM algorithm [3] that we have implemented, the ownership of a page does not change with read-requests. The owner of a page keeps a copy-set of all clusters that have valid copies of the page, and invalidate them when a write-request is received. All cluster being invalidated will then send an acknowledgment message back to the requesting cluster, which must wait for all the

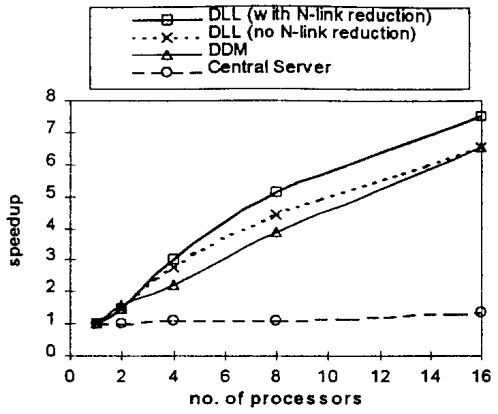

Figure 6: Speed-up of linear equations solver

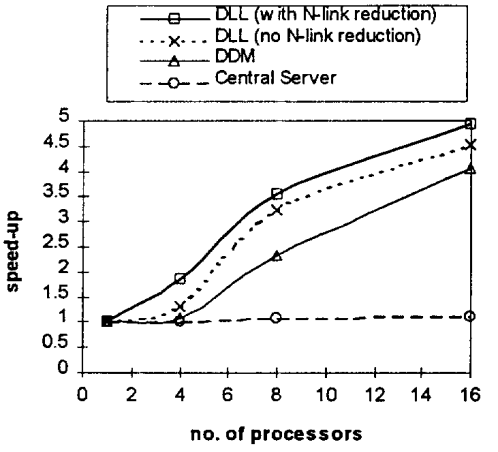

Figure 7: Speed-up of matrix multiplier

acknowledgment messages to arrive before the write-request can be completed.

All the algorithms are implemented as user level programs in a network of workstations running PVM 3 [9]. The network transfer rate of 0.8 byte/cycle (equivalent to 40MB/s on a 50MHz system) and the message passing latency of 500 cycles are assumed [8, 10, 11]. The page size is set to be 1k Byte for all three algorithm. In various studies of interconnection network performance, the latency is shown to rise sharply when the network becomes saturated [10, 11, 12].

Two common problems are to be solved in the simulation. First, a set 256 linear equations is solved using the Gauss-Seidel method [7]. Second, two square matrix of size 48×48 are being multiplied together using a parallel matrix multiplier. Systems of up to 16 processors have been simulated and the speed-up obtained by various algorithms are summarized in *Figure 6 and 7*. The speed-up is calculated by:

$$\text{speed - up} = \frac{\text{time required by a single processor}}{\text{time required by n processor}}$$

As can be seen from the graphs, all the DSM algorithms, with the exception of the Central Server algorithm, achieve considerable speed-up even with only a moderately large problem size. For the Central Server algorithm, the speed-up obtained by using 16 processors is just around 1.3 for both problems. This is due to the system bottleneck at the cluster that acts as the central server. As all remote memory accesses must go through the central server, serious network congestion occurs and the server becomes a serializing point for the whole system, thus defeating the purpose of parallel processing.

For the other three algorithms, the DLL protocol with N-link reduction generally has the best performance. For the linear equation solver, the speed-up improvement over the DDM algorithm is 32.38% and 14.71% for 8 and 16 processors, respectively. For the matrix multiplier, the improvement is 51.94% and 21.84%.

The DLL protocol without N-link reduction performs better than the DDM algorithm when the number of processors used is less than or equal to 8. However, as the number of processors grows to 16, its performance dropped and becomes closer to that of the DDM algorithm. This is because when the number of processors in the system grows larger, the chains of N-links can become very long and it can take a long time for memory access requests messages to reach the corresponding owner of the page.

On the other hand, the DDM algorithm also achieves considerable speed-up, though not as good as the speed-up achieves by the two DLL protocols. This is due to the large number of messages generated by the DDM algorithm in memory write accesses when the owner sent a burst of messages to invalidate copies of the page in other clusters.

Note that from the figures, the speed-up improvement (shown by the gradient of the curve) of the DLL protocol is greater when the number of processors increases from 1 to 8, and is smaller when the number of processors increases beyond 8. The main reason for this is that inter-processor communication overhead increase with the number of processors and for the moderate size problems that we

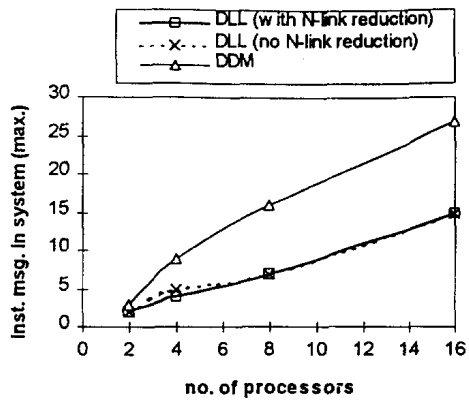

Figure 8: Maximum instantaneous number of messages in system for linear equations solver

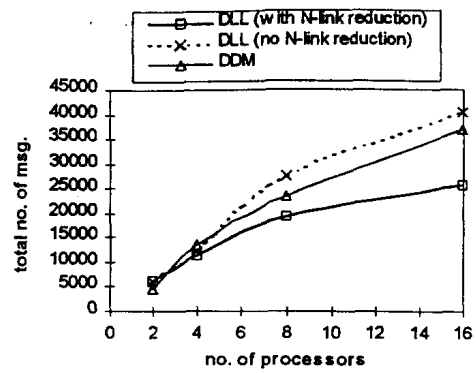

Figure 10: Total number of messages required to solve the set of 256 linear equations

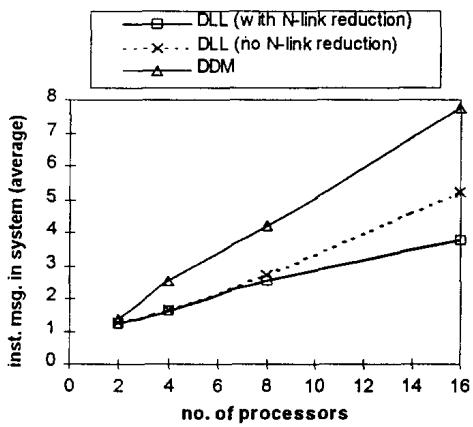

Figure 9: Average instantaneous number of messages in system for linear equations solver

are solving, the overhead become significant as the number of processors used is more than 8. If larger problems are to be solved, better speed-up improvement will be achieved by adding more processors.

In order to have a clearer picture of the number of messages used by the DLL and the DDM algorithms, the maximum instantaneous number of messages in system (*Figure 8*) and the average instantaneous number of messages in system (*Figure 9*) is plotted against the number of processors for the linear equations solver. Similar results have been obtained from the matrix multiplier. Note that the Central Server algorithm is not included in the comparison because its performance is too poor and it would be unfair to compare the number of messages it generates with the other algorithms in concern.

As seen from *Figure 8 and 9*, the DDM algorithm has both the highest maximum instantaneous number and the highest average instantaneous number of messages in system. This is due to the frequent burst of invalidation messages generated by the DDM algorithm from memory write access. The constant high number of messages in the system means that the DDM algorithm is more prone to the network congestion problem. This problem will become more significant when the problem size is large, because then more pages will be required to store the data and results, thus more clusters will need to perform invalidation requests simultaneously.

On the other hand, since the DLL protocol uses the P-links to one-by-one invalidate other clusters, the maximum instantaneous number of messages in the system is kept below the number of processors. Therefore, network congestion seldom occurs. This is true even when the problem size increases because according to the DLL protocol, messages are generated one at a time by each cluster, as oppose to the burst of messages generated by the DDM algorithm during invalidation. Hence, it is expected that the DLL protocol scales better with increasing problem size. Also, note that the average number of messages in system is smaller when N-link reduction is used, owing to the smaller number of messages needed to locate the owner of a page.

Figure 10 is the plot of the total number of messages used to solve the set of 256 linear equations against the number of processors for the DLL and the DDM algorithms. From the graph, the DLL protocol without N-link reduction requires the highest number of messages to solve the equations. By comparing this

with the number of messages required by the DLL protocol with N-link reduction, we can see that a large number of messages were actually wasted going through the N-link locating the owner of pages. This shows the importance of the N-link reduction feature to the DLL protocol.

Also from the graph, the number of messages required by *DDM* algorithm to solve the equations is significantly larger than the DLL protocol with N-link reduction. The extra messages are actually the extra acknowledgment messages used by the DDM algorithm in the invalidation process. In the DDM algorithm, to invalidate copies of a page in n different clusters, 2n messages are needed (n invalidation messages and n acknowledgment messages). In the DLL protocol, however, only n+1 messages are needed (n invalidation messages and 1 acknowledgment message). Therefore, the more processors in the system, the more messages the DLL protocol will save.

5. Further Research on the DLL Protocol

From the simulation results, we can see that the DLL protocol provides a promising and feasible solution to the DSM memory coherence problem. The possibility of implementing the DLL protocol into a wide range of systems calls for further investigations of the protocol.

First, as the DLL protocol was originally designed for use in distributed operating systems running on MPP, it is desirable that the protocol be actually implemented and thus performance evaluation using the real implementation, rather than just simulation, is possible. In fact, our group is current building a MPP system using the hierarchical cluster model. The system will run a version of MACH [13] with build-in DSM using the DLL protocol.

Second, from the experience of implementing the DLL protocol into a network of workstations running PVM for the simulation purpose, we found that it might be feasible to include the protocol as an add-on library for PVM, or as an alternative library so that users can access the distributed memory of the workstations transparently as a global shared memory. As the costs of powerful workstations drop, this could offer a new, cost effective, and yet user-friendly way of parallel processing.

Third, although the current implementation of the DLL protocol uses the sequential consistency model, it is possible to convert the protocol to use a relaxed consistency model [2] with only minor modifications. Moreover, it is expected that the DLL protocol will be suitable for relax coherence models because then a cluster does not need to wait for the invalidation message to go through a chain of P-links before it can complete a normal write operation. In addition, the use of the write-update algorithm instead of write-invalidate algorithm for normal write operations will then be possible, although write-invalidate should still be used for synchronized write accesses because of its shorter completion time.

6. Conclusion

This paper has introduced the Doubly-Linked List protocol for Distributed Shared Memory systems. Detailed explanations of the protocol as well as performance evaluation and comparison with other existing protocols using simulation studies have also been presented. By using the linked list of clusters, the DLL protocol provides a fully distributed way to keep track of replicated pages in the system, thus eliminating the network congestion problem caused by the generation of a large number of messages within a short time by a single cluster. It is shown that the DLL protocol provides a high performance and scalable solution to implementing DSM in multiprocessor systems.

7. Acknowledgment

This project is supported in part by the University of Hong Kong CRGC Grant 337/062/0012.

References

[1] B.Nitzberg and V.Lo, "Distributed Shared Memory: A Survey of Issue and Algorithms," *Computer, IEEE*, pp. 52-60, Aug. 1991.

[2] K. Hwang, Advanced Computer Architecture, McGraw Hill, pp. 19-27, pp. 248-256, pp. 487-590, pp. 1993.

[3] K.Li and P.Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7 No. 4, pp. 321-359, Nov. 1989.

[4] K.Li and R.Schaefer, "A Hypercube Shared Virtual Memory System," *Proceeding of 1989 International Conference on Parallel Processing*, pp. I-125 - I-132, Aug. 1989.

[5] M.Stumm and S.Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer, IEEE*, pp. 54-64, May. 1990.

[6] R.Bisiani and M.Ravishankar, "Plus: A Distributed Shared-Memory System," *Proceeding of 17th International Symposium on Computer Architecture*, pp. 115-124, 1990.

[7] S.Akl, *The Design and Analysis of Parallel Algorithms, Prentice Hall*, 1989, pp. 203-205.

[8] TMS320C4x User's Guide, *Texas Instruments*, pp.1-1 - 1-12, 1992.

[9] A.Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek, V.Sunderam, PVM 3 User's Guide and Reference Manual, *Oak Ridge National Laboratory*, 1994.

[10] X.Lin, P.K.McKinley, L.M.Ni, "Deadlock-Free Multicast Wormhole Routing in 2-D Mesh Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, pp. 793-804, Aug. 1994.

[11] J.Kim, A.Chien, "The Impact of Packetization in Wormhole-Routed Networks," *Proceedings of Parallel Architectures and Languages Europe*, June 1993.

[12] W.Dally, "Virtual-Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems*, pp. 194-205, Mar. 1992.

[13] M.Accetta, R.Baron, W.Bolosky, D.Golub, R.Rashid, A.Tevanian and M.Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of Summer 1986 USENIX Conference*, pp. 93-113, June 1986.