Short Paper

Incorporating Memory Resource Considerations into the Workload Distribution of Software DSM Systems

YEN-TSO LIU[1], TYNG-YEU LIANG[2], JYH-BIAU CHANG[3] AND CE-KUEN SHIEH[1]
[1]*Department of Electrical Engineering*
National Cheng Kung University
Tainan, 701 Taiwan
[2]*Department of Electrical Engineering*
National Kaohsiung University of Applied Sciences
Kaohsiung, 807 Taiwan
[3]*Department of Information Management*
Leader University
Tainan, 709 Taiwan
E-mail: {andy[1]; lty[2]; andrew[3]; shieh[1]}@hpds.ee.ncku.edu.tw

Conventional workload distribution schemes for software distributed shared memory (DSM) systems simply distribute the program threads in accordance with the CPU power of the individual processors or the data-sharing characteristics of the application. Although these schemes aim to minimize the program execution time by reducing the computation and communication costs, memory access costs also have a major influence on the overall program performance. If a processor has insufficient physical memory space to cache all of the data required by its local working threads, it must perform a series of page replacements if it is to complete its thread executions. Although these page replacements enable the threads to complete their tasks, thread execution is inevitably delayed by the latency of the page swapping operations. Consequently, the current study proposes a novel workload distribution scheme for DSM systems which considers not only the CPU power and data-sharing characteristics, but also the physical memory capabilities of the individual processors. The present results confirm the importance of considering memory resources when establishing an appropriate workload distribution for DSM systems and indicate that the proposed scheme is more effective than schemes which consider only CPU resources or memory resources, respectively.

Keywords: workload distribution, distributed shared memory (DSM), memory resource, page replacement, data-sharing characteristic

1. INTRODUCTION

Obtaining a suitable workload distribution is essential if the performance of programs executed in parallel on clusters of computers is to be optimized. Most users generally partition their problems evenly into a number of threads and then distribute these threads equally onto each processor in an attempt to achieve a workload balance. However, the

Received October 28, 2004; revised March 11 & June 8 & October 18, 2005; accepted December 19, 2005.
Communicated by Chu-Sing Yang.

processors clustered in computer networks are generally not identical in terms of their resource capabilities, *e.g.* CPU power, physical memory space, network bandwidth, *etc.* Consequently, some processors may possess insufficient resources to match the demands of their local threads, while others may have insufficient work to fully exploit their available resources. This not only delays the overall finish time of the program, but also reduces the return obtained from the original investment in the computational resources. Therefore, it is necessary to develop a more sophisticated scheme to distribute program workloads onto clustered computers in accordance with their specific resource capabilities.

Recently, software distributed shared memory (DSM) systems have been successfully applied for the parallel scheduling of program workloads onto clustered computers. These run time systems use software technology to construct a virtual shared memory abstract over the clustered computers. With this virtual shared memory abstract, users can employ shared variables rather than message passing to develop their applications. This greatly reduces the complexity of programming applications designed for execution on clustered computers. However, DSM systems suffer the same workload distribution problems as those described in the paragraph above. Although, some DSM systems employ dynamic workload distribution schemes to optimize the performance of user applications [1-4], these schemes consider only the CPU power and the data sharing characteristics of the application when distributing the program threads onto the clustered processors.

Besides computation and communication costs, memory access costs also play a key role in determining the overall program performance. If a processor has insufficient physical memory space to cache all of the data demanded by its local threads, it will be required to perform page replacements at run-time whenever its local threads attempt to access data which is not located in its physical memory. Although virtual memory technology enables the processor to complete the tasks of its local threads, the memory swapping routines inevitably postpone the thread execution. The rapid advances made in VLSI technology over the past decade have increased the relative influence of memory swapping delays on the program performance since the speed differential between the CPU and external storage devices is greater than that between the CPU and its physical memory. Therefore, workload distribution methods which take no account of memory resources are liable to make flawed decisions which degrade rather than enhance the performance of DSM programs.

In order to resolve this problem, this paper discusses the inclusion of memory resource considerations in the workload distribution planning of software DSM systems. In the first step of the present research, the memory resources of the processors are taken into account in deriving a set of formulae with which to predict the execution time of each processor for a specific thread-mapping pattern when executing users' DSM programs. These formulae provide the means to identify the thread-mapping pattern which improves the system performance by simultaneously considering the computational and memory resources of each processor and the data sharing characteristics of the DSM application. Having identified the optimum thread-mapping pattern, the current thread-mapping pattern is adjusted at run-time in order to reduce the delay latencies between processors, hence improving the performance of the DSM application. In the second stage of the current study, the proposed workload distribution scheme is implemented on a testbed known as Teamster [9] in order to examine its effect on the performance of various test applications.

The remainder of this paper is organized as follows. Section 2 discusses previous studies of workload distribution on DSM systems. Section 3 analyzes the impact of memory resource considerations on the performance of DSM applications and develops analytical formulae to establish the iteration finish times of the executing processors under different thread mapping patterns. Section 4 discusses the architecture of the proposed workload distribution scheme on Teamster. Section 5 presents the results of a series of experiments designed to investigate the effect of considering memory resources on the application performance when planning the workload distribution of DSM systems. Finally, section 6 draws some brief conclusions and highlights the areas of intended future research.

2. RELATED WORKS

Current DSM systems capable of supporting workload balancing via dynamic workload redistribution include CVM [5], JIAJIA [6] and Cohesion [7]. CVM focuses on obtaining the workload balance which minimizes the communication costs incurred when maintaining data consistency. CVM achieves this by distributing the program threads onto processors in accordance with the computational powers of the individual processors and the computational demands of the working threads. Furthermore, CVM co-locates pairs of threads having the highest degree of mutual data sharing on the same node in order to minimize inter-node communication costs. JIAJIA assumes that each processor has sufficient physical memory space to hold the data required by the threads assigned to it. Adopting a similar approach to that taken by CVM, JIAJIA distributes the program workload based on the computational powers of the individual processors. The third system, Cohesion, divides the program workload distribution task into two phases, namely the migration phase and the exchange phase. In the migration phase, Cohesion estimates the appropriate workload for each processor using the same approach as that employed by CVM and then migrates threads from the heavily loaded nodes to the more lightly loaded nodes in order to minimize load imbalance costs. Meanwhile, in the exchange phase, pairs of threads with the highest degree of mutual data sharing are co-located on the same node in order to reduce the communication costs incurred by thread exchanges.

The three systems described above all neglect the memory resources of the individual processors in the computer cluster when distributing the application workload. Some researchers [12] have considered both processor and memory factors when developing the workload distribution policy for applications designed for execution on distributed systems. In the proposed study, the system adopts a CPU-memory-based allocation policy if the memory resources are insufficient. To optimize the application performance, the proposed policy allocates jobs to a node only if it has sufficient memory resources to process the task. However, this approach is not suitable for DSM systems. In the parallel computation of a DSM application, the main task is generally partitioned into several subtasks and dispatched to a number of different nodes. The DSM system then collects the execution results of the individual subtasks to obtain the result of the main task. Accordingly, the extent of the parallelism between each subtask is significant interest when developing workload distribution schemes for software DSM systems.

3. FORMULA ANALYSIS

DSM applications can be categorized into three broad groups, namely *fork-join*, *run-to-complete* and *iterative* [8]. Since each group has different execution characteristics, it is necessary to design specific workload distribution schemes for each one. The current workload distribution investigation focuses on iterative DSM applications. Compared to *fork-join* and *run-to-complete* applications, iterative applications more readily provide the information required to derive precise estimates of the program execution time for different thread-mapping patterns. Furthermore, iterative applications provide a clearer view of the impact of different workload distributions on the program performance.

In software DSM systems, an iterative program is generally partitioned into a number of threads, which are then distributed onto processors for parallel execution. When individual threads finish their jobs within the current iteration, they must join the other threads at a barrier before commencing the next iteration. Accordingly, the finish time of any iteration is determined by the longest finish time of any processor working within that iteration. Clearly, the total execution time of the iterative program is given by the sum of the finish times of the individual iterations created in that program. Therefore, to evaluate the effect of the workload distribution on the application performance, this study develops a formula to estimate the iteration finish time of a processor with a specific thread-mapping pattern.

Basically, the iteration finish time of processor x, T^x, comprises three components, the *computation time*, the *memory swapping time* and the *communication time*, *i.e.* $T^x = T^x_{comp} + T^x_{mem} + T^x_{comm}$.

The computation time, T^x_{comp}, is the time spent by processor x executing the computational work of its local threads. Let S_x be the set of all threads running on processor x and t^i_{comp} be the computation time of thread i assigned to processor x. T^x_{comp} is therefore given by $T^x_{comp} = \sum_{i \in S_x} t^i_{comp}$.

The communication time, T^x_{comm}, is the time spent by processor x propagating its data page updates to other processors holding the same page in order to maintain data consistency. According to Liang [11], T^x_{comm} is given by $T^x_{comm} = \sum_{y=1, y \neq x}^{N} \sum_{k=1}^{P} C_{xyk}$, where $C_{xyk} = \left\lceil \dfrac{\phi\left(\bigcup_{i \in S_x} diff_{ik}\right)}{Size_{packet}} \right\rceil \times t_{packet}$ if processor x and processor y share page k, and $C_{xyk} = 0$ otherwise. In this equation, N is the number of execution processors, P is the total number of data pages, $Size_{packet}$ is the network packet size, and t_{packet} is the average time spent transferring one message packet in the network. If processor x shares page k with processor y, processor x will send its updates for page k to processor y to maintain data consistency. The update of processor x for page k is actually the accumulation of its local threads' updates for page k. Consequently, processor x will send the update of page k, *i.e.*,

$\bigcup_{i \in S_x} diff_{ik}$, to processor y if the threads located on processor x have written page k. The number of message packets for sending $\bigcup_{i \in S_x} diff_{ik}$ is then given by $\left\lceil \frac{\phi\left(\bigcup_{i \in S_x} diff_{ik}\right)}{Size_{packet}} \right\rceil$, where $\phi\left(\bigcup_{i \in S_x} diff_{ik}\right)$ is the size of $\bigcup_{i \in S_x} diff_{ik}$.

Finally, the memory swapping time, T^x_{mem}, is the time spent by processor x performing page replacements to cache the data accessed by its local threads. Let M_x represent the maximum memory space which processor x can afford for thread memory demands. Furthermore, let m_i be the memory space requested by thread i. In the case where a processor has sufficient memory space to cache all of the data accessed by its local threads, the latency of memory accesses can be neglected. However, if the processor has insufficient memory space, page faults will occur frequently during thread execution as virtual memory mechanisms are triggered to perform page replacements. The memory accesses of the threads will be delayed due to the latency of executing these page replacements. Consequently, the memory swapping time of processor x can be denoted as:

$$T^x_{mem} = \begin{cases} \sum_{i \in S_x} t^i_{mem}, & \text{if } \bigcup_{i \in S_x} m_i > M_x \\ 0, & \text{if } \bigcup_{i \in S_x} m_i \le M_x \end{cases}$$

where t^i_{mem} is the memory swapping latency of thread i assigned to processor x.

The time required to execute a page replacement can be divided into two discrete components. The first component is the time spent scanning the physical memory to identify the least-recently used (LRU) data pages and then swapping these pages out to disk. The second component relates to the time spent swapping in the data pages required by the threads from disk to the physical memory. Therefore, the memory swapping latency of thread i assigned to processor x can be expressed as $t^i_{mem} = f^i \times \left(t^x_{spi} + t^x_{spo}\right)$, where f^i is the number of page replacements executed by processor x in caching the data pages required by thread i, t^x_{spo} is the average time spent by processor x searching for the LRU data pages and swapping one page out to the swapping device, and t^x_{spi} is the average time spent swapping in one page on processor x. Therefore, T^x_{mem} can be divided into two discrete components, *i.e.* the memory swapping-in time, T^x_{spi}, and the memory swapping-out time, T^x_{spo}. T^x_{spi} means the total time spent by the system swapping in pages for the DSM application on processor x. T^x_{spo} is the total time spent by the system searching for the LRU data pages on processor x and then swapping these pages out.

In general, the number of swapping-out operations is equal to the number of swapping-in operations, and the occurrence of these operations varies as a function of the memory deficiency. However, most UNIX-like operating systems use an individual page scanning [13] process to scan the entire physical memory space in order to identify the LRU pages which can be swapped out. The performance of this page scanner is directly proportional to the processor power. Additionally, the probability of identifying LRU pages for page replacement is directly related to the size of the available physical memory which the system can offer the DSM application. In other words, T^x_{spo} is inversely proportional to the size of the available physical memory. Assuming that each processor in the DSM system has similar I/O hardware architectures, the memory swapping-in and swapping-out time equations can be simplified to:

$$T_{spi}^{x} = \sum_{i \in S_x} f^{i} \times t_{spi}^{x} \propto M_{lack}^{x}$$

$$T_{spo}^{x} = \sum_{i \in S_x} f^{i} \times t_{spo}^{x} \propto \frac{M_{total}^{x} \times M_{lack}^{x}}{M_{free}^{x} \times C^{x}}$$

where C^x indicates the power factor (*i.e.* speed) of processor x, M_{lack}^{x} denotes the volume of the physical memory deficiency at processor x while running the specific DSM application, M_{total}^{x} represents the total size of the physical memory at processor x, and M_{free}^{x} indicates the available physical memory for the DSM application at processor x.

When implementing the proposed workload distribution scheme, the processor which provides the most comprehensive system information, particularly the memory swapping time, is specified as the reference processor. Run-time information collected from this processor is then used by the system to estimate the execution time of the DSM application for different thread-mapping patterns. The iteration finish time of processor x is expressed in the following final form:

$$T^{x} = \sum_{i \in S_x} t_{comp}^{i} + \left(\frac{M_{lack}^{x}}{M_{lack}^{ref}} \times T_{spi}^{ref} + \frac{M_{lack}^{x}}{M_{lack}^{ref}} \times \frac{M_{total}^{x}}{M_{total}^{ref}} \times \frac{M_{free}^{ref}}{M_{free}^{x}} \times \frac{C^{ref}}{C^{x}} \times T_{spo}^{ref} \right) + T_{comm}^{x}$$

where T_{spi}^{ref} is the total time spent by the system swapping in pages for the DSM application on the reference processor, T_{spo}^{ref} is the total time spent by the system searching for the LRU data pages on the reference processor and then swapping these pages out, C^{ref} is the power factor of the reference processor, M_{lack}^{ref} is the physical memory deficiency of the reference processor, M_{total}^{ref} is the total size of the physical memory of the reference processor, and M_{free}^{ref} is the physical memory available to the DSM application at the reference processor.

4. PROPOSED WORKLOAD DISTRIBUTION SCHEME

The proposed dynamic workload distribution scheme is designed to adjust the thread-mapping pattern at run-time such that the delay latencies between processors are minimized, hence improving the performance of the DSM application. As shown in Fig. 1, the proposed workload distribution scheme comprises three phases, the information collection phase, the prediction phase, and the reconfiguration phase. During the first phase, run-time information is collected and maintained as an input to the trigger manager and the prediction phase. The trigger manager is used to determine whether the DSM system is suffering from a workload imbalance. When prompted by the trigger manager, the prediction phase tries to generate a new thread-mapping pattern to improve the performance of the DSM application. The reconfiguration phase then redistributes the

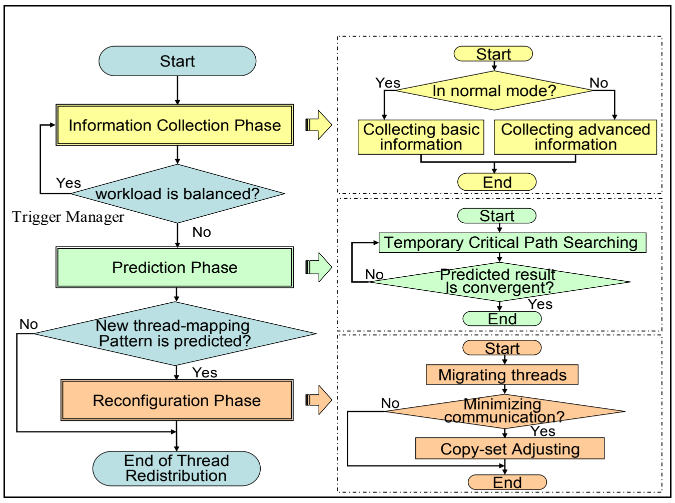
Start
Information Collection Phase
Yes
workload is balanced?
Trigger Manager
No
Prediction Phase
No
New thread-mapping Pattern is predicted?
Yes
Reconfiguration Phase
End of Thread Redistribution
Start
Yes
In normal mode?
No
Collecting basic information
Collecting advanced information
End
Start
Temporary Critical Path Searching
No
Predicted result Is convergent?
Yes
End
Start
Migrating threads
No
Minimizing communication?
Yes
Copy-set Adjusting
End

Fig. 1. Flowchart of proposed workload distribution scheme.

working threads in accordance with the new thread-mapping pattern identified by the prediction phase. The details of these three phases are discussed in the paragraphs below.

4.1 Information Collection Phase

The information required by the proposed workload distribution method can be classified as either *static* or *dynamic*. The former includes the CPU power, the total physical memory space, and the average time spent executing page replacements. This static information is collected only when the execution of the DSM application is initiated. The dynamic information includes the computation time of each working thread, the data access pattern of the working threads, and the residual memory space of each processor. This information is collected during the execution of the DSM application.

Since the current testbed, Teamster, is built at the user level, it is easily modified to support the collection of the required dynamic information. Tools such as “SE” can be used to gather some system information, *e.g.* the residual physical memory space, from the *procfs* file systems, while the *active correlation track* mechanism [10] can be employed to collect the data access pattern of the working threads. The data access pattern is used to identify the working sets of threads and to determine the amount of memory required by these threads. It can also be used to estimate the communication time of each processor for a specific thread-mapping pattern.

The data access pattern plays a particularly important role in the proposed workload distribution scheme. However, the overheads involved in identifying this data access pattern are relatively higher than those incurred in collecting the other system information. Hence, the information collection phase shown in Fig. 1 is implemented as a two-subphase module comprising a basic information collection module and an advanced information collection module. Initially, the system operates in a normal mode and the basic information collection module acquires routine system information at run-time. However, when an obvious workload imbalance situation is identified, the advanced information collection module is activated to collect more complex information such as the data access pattern. Using this two-stage approach, the proposed information collection phase achieves the goal of collecting the information required by the subsequent prediction phase in an efficient and cost effective manner.

4.2 Prediction Phase

In Fig. 1, the prediction phase applies the formulae derived in section 3 to predict the thread-mapping pattern which will effectively improve the performance of the DSM application. However, using a full-search algorithm to identify the optimal thread-mapping pattern incurs excessive overheads. Therefore, this study makes the fundamental assumption that the behaviors of the iterative DSM applications considered in the present investigation are regular, *i.e.* the processor and memory resource demands of each thread are similar within the same application. On this basis, an efficient heuristic algorithm is developed to establish the optimum thread-mapping pattern.

The prediction phase is divided into two sequential subphases. The objective of the first subphase is to predict the number of threads which should be assigned to each processor in order to achieve a workload balance, thereby minimizing the waiting latencies when executing the iterations of the DSM application. This study incorporates memory resource considerations into this subphase by performing the following steps:

(1) Identify the processor with the longest finish time and the processor with the shortest finish time and designate these processors as the source processor and the destination processor, respectively. Define the partial iteration time as the longest finish time between these two processors.
(2) Simulate the migration of one thread from the source processor to the destination processor. Use the final formula presented in section 3 to estimate the finish times of these two processors and determine the partial iteration time between them. Assess whether or not the estimated partial iteration time is reduced by the simulated migration. If this is the case, repeat step (2) until the partial iteration time between the two processors cannot be reduced any further. (Note that this iterative search procedure is illustrated in Fig. 2.)
(3) If the simulations in step (2) indicate that one or more threads should be migrated from the source processor to the destination processor, update the number of threads assigned to each processor, re-estimate their respective finish times, and then return to step (1).
(4) If step (2) does not identify any threads for migration, identify the processor which has the shortest finish time other than the previously designated destination processors which do not lead to any further thread migrations from the current source processor, specify this processor as the new destination processor, and repeat step (2).
(5) When no other processors exist to be specified as the new destination processor, *i.e.* the steps above have been performed using each of the processors other than the source processor as the destination processor, the task of predicting the number of threads to be assigned to each processor is complete.

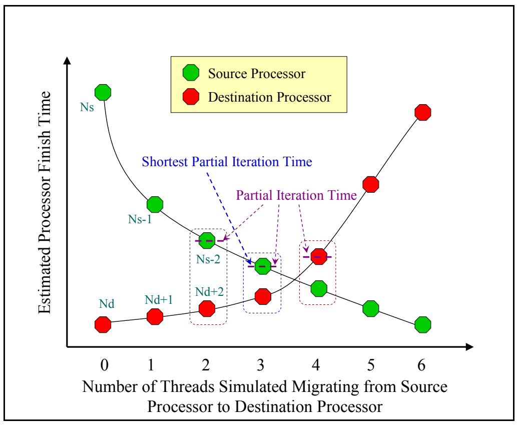

Fig. 2. Search method for shortest partial iteration time.

Having completed the simulation activities described in steps (1) to (4) above, the system produces a complete description of the predicted thread number pattern, expressed in terms of the number of threads to be assigned to each processor. The second prediction subphase uses the results of the first subphase and the data access pattern information provided by the information collection phase to predict the thread-mapping pattern which will minimize the data sharing costs between the executing processors. If the predicted thread-mapping pattern differs from the current thread-mapping pattern, the result is passed to the reconfiguration phase, which then executes the physical redistribution of the working threads, as described below.

4.3 Reconfiguration Phase

The reconfiguration phase is also divided into two sequential subphases. In the first subphase, referred to as the migration subphase, the system utilizes a thread migration mechanism to adjust the distribution of the working threads according to the results of the preceding prediction phase. Having completed the migration subphase, the DSM application is prompted to proceed to the next iteration. Most of the threads in iterative DSM applications tend to access the same data pages repeatedly. The multiple writer protocol of the release consistency in the Teamster testbed leads to the generation of a large number of communication messages relating to the data consistency maintenance of pages rendered redundant because of the thread migration operations. Therefore, the second reconfiguration subphase, referred to as the communication minimization subphase, deletes the data pages no longer required from the threads on the original processor and removes the corresponding copysets from the page owners. Consequently, the volume of redundant messages generated by the dynamic workload distribution scheme is substantially reduced.

5. EXPERIMENTS

Teamster is a user-level DSM system built on a cluster of Intel 80x86 PCs, running the Sun Solaris 8 operating system, and connected by a 100Mbps Fast Ethernet network.

Teamster removes the need for data address translation between processors by providing a single, global address space for user applications. Furthermore, Teamster supports multiple memory consistency protocols, *i.e. sequential* and *eager released*, in order to reduce the communication costs associated with maintaining data consistency.

This study implements seven iterative applications, *i.e.* SOR, Matrix Multiplication (MM), Gaussian Elimination (GE), Jacobi, N-Body, MPEG4 Encoder (MPEG4) and Vector Quantization (VQ), to evaluate the effect of including memory resource considerations in the workload distribution decision-making process. Each application creates 32 threads to handle its tasks. The SOR application is used in the modeling of natural phenomena, *e.g.* for determining temperature gradients over a square area given the temperature values at the area boundaries. The MM application computes the results of $C = A * B$, where A, B, and C are N-by-N square matrices. The Gauss Elimination (GE) application solves sets of simultaneous equations by eliminating variables from successive equations. The Jacobi application is a well-known mathematical method commonly used in the scientific engineering domain to solve the two-dimensional, first-order finite difference equation resulting from the Laplace equation. N-Body is an astrophysics application used to calculate the forces among particles. This application calculates the total force on every particle and updates the particles' positions and other attributes in a self-gravitating space system according to Newton's acceleration theorem. The MPEG4 application uses a full-search method to establish the motion vector of each frame in a sequence. Finally, VQ is a well-known image compression technique.

5.1 Experimental Environments

Table 1 presents the details of the computer clusters used to execute the seven test applications. The first experimental group comprises two environments, in which the four nodes have identical CPU power processors, but each node has a different amount of available physical memory. In the second experimental group, the physical memory size of each individual node is constant, but each node has a different CPU power processor. In the third experimental group both the CPU power and the available physical memory size of the individual nodes may vary. Note that in all experimental groups, the volume of available physical memory at each node is controlled using a tool with negligible costs.

Table 1. Organization of computer clusters used for executing test applications.

Group 1	Node 0 (Mhz - MB)	Node 1 (Mhz - MB)	Node 2 (Mhz - MB)	Node 3 (Mhz - MB)
A	500 - 100	500 - 85	500 - 70	500 - 55
B	500 - 90	500 - 75	500 - 60	500 - 45
Group 2	Node 0 (Mhz - MB)	Node 1 (Mhz - MB)	Node 2 (Mhz - MB)	Node 3 (Mhz - MB)
A	500 - 100	400 - 100	400 - 100	300 - 100
B	500 - 80	400 - 80	400 - 80	300 - 80
Group 3	Node 0 (Mhz - MB)	Node 1 (Mhz - MB)	Node 2 (Mhz - MB)	Node 3 (Mhz - MB)
A	500 - 90	400 - 75	400 - 60	300 - 45
B	500 - 45	400 - 60	400 - 75	300 - 90

Each application is executed with four different policies in the first subphase of the prediction phase, *i.e.* an initial policy, a CPU-based policy, a MEM-based policy, and a CPU&MEM-based policy. The initial policy distributes the application threads equally onto the cluster nodes. The CPU-based policy considers only the CPU resource, and ignores the memory resource. The MEM-based policy distributes threads according to the amount of physical memory space each processor is able to provide. Finally, the CPU& MEM-based policy, which is the default policy of the proposed workload distribution scheme, considers both CPU resources and memory resources simultaneously.

5.3 Experimental Results

Fig. 3 shows the iteration times of the experimental applications with each workload distribution policy for the two test environments within Experimental Group 1. The four bars within each cluster correspond to the four different policies and the height of each bar represents the iteration time of that particular application. For a long-term iterative DSM application, a performance improvement is defined as a reduction in the average execution time of the iterations. Therefore, this study uses the iteration time rather than the execution time to assess the program performance improvement since the tested DSM applications are all iterative and exhibit a regular behavior.

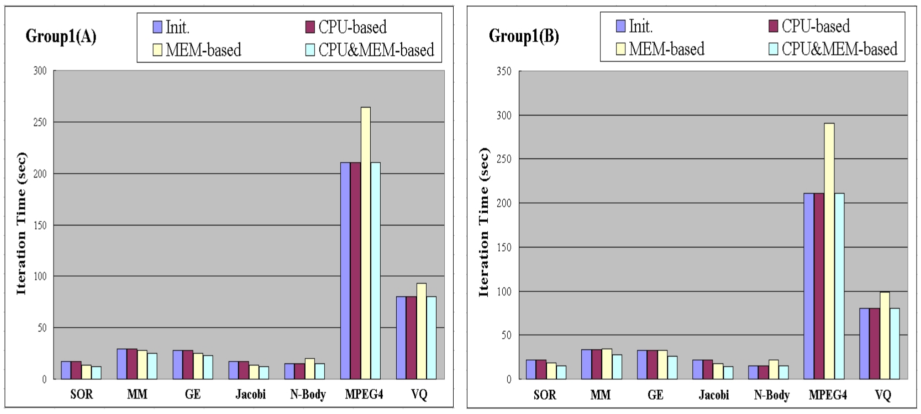

Fig. 3. Experimental results for group 1.

From Fig. 3, it is clear that the CPU-based policy is ineffective in reducing the iteration time, *i.e.* there is no difference between the iteration time obtained when the CPU-based policy is applied and that achieved when the initial policy is used. However, both the MEM-based policy and the CPU&MEM-based policy achieve significant reductions in the iteration times of the SOR, MM, GE and Jacobi applications, which require huge amounts of memory resources. When executing these applications, some of the cluster nodes within the Experimental Group 1 environments have insufficient physical memory space to carry out their tasks. Therefore, they suffer from memory swapping delays when the initial or CPU-based policies are applied and these delays increase the iteration time significantly. However, in all of the trials performed in the Experimental Group 1 environments other than the MM application in Group 1(B), both the MEM- based policy and the CPU&MEM-based policy provide consistently superior results for the SOR, MM, GE and Jacobi applications. Additionally, it is noted that the CPU-based policy outperforms the MEM-based policy when applied to the MM application in the Group 1(B) test environment. This observation is explained by the fact that the influence of the memory swapping delays is less than that of the CPU load imbalance for the MM application in Group 1(B).

In Fig. 3, the results of the last three applications, *i.e.* N-Body, MPEG4 and VQ, are different from those of the first four applications, *i.e.* the performance of the CPU-based policy for these three applications rivals that of the CPU&MEM-based policy. The reason for this is that the memory demands of these three applications are quite small and hence every node in the experimental environment has sufficient available memory resources to meet the demands of its local working threads regardless of the workload distribution policy applied. Consequently, the MEM-based policy generates a load imbalance situation and therefore reduces the performance of the three applications. However, the CPU&MEM-based policy considers both the memory resources and the computation resources of each node and therefore produces the best results in each of the Experimental Group 1 environments.

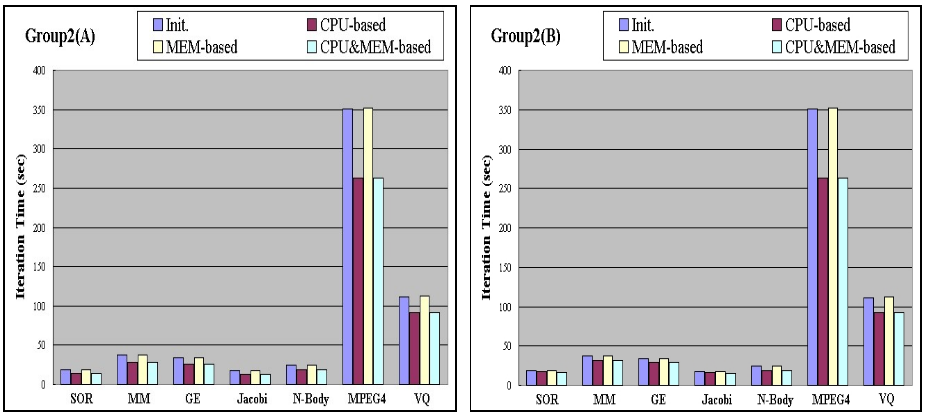

Fig. 4. Experimental results for group 2.

Fig. 4 shows the iteration times of the experimental applications with each workload distribution policy for the two test environments within Experimental Group 2. In general, it can be seen that the CPU-based policy is more effective than the MEM-based policy in both the first (A) and the second (B) test environments. However, in Experimental Group 2(B), the performance of the Jacobi application suffers when the CPU-based policy is applied in place of the MEM-based policy. This observation is explained by the significant memory swapping delays caused by the CPU-based policy. Specifically, the influence of the memory swapping delay is greater than that of the CPU load imbalance. Fig. 4 also demonstrates that the proposed CPU&MEM-based policy consistently provides a superior performance of the SOR, MM, GE and Jacobi applications within these two environments.

As in the case of Experimental Group 1, the results of the N-Body, MPEG4 and VQ applications differ from those of the SOR, MM, GE and Jacobi applications in both Experimental Group 2 environments. This is again explained by the relatively small memory demands of these three applications, which ensures that each node has sufficient memory resources to accommodate the demands of its local working threads irrespective of the workload distribution policy applied. Consequently, the MEM-based policy, which does not consider CPU resources, fails to improve the system performance of these three applications. It is observed that the proposed CPU&MEM-based policy provides the same results as the CPU-based policy for the N-Body, MPEG4 and VQ applications when the memory resources of each node satisfy the memory demands of the local working threads.

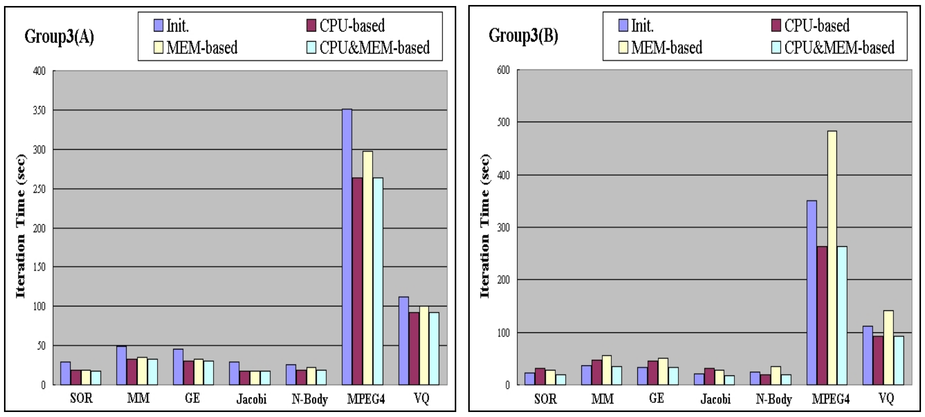

Fig. 5. Experimental results for group 3.

Fig. 5 shows the iteration times of the experimental applications with each workload distribution policy for the two test environments within Experimental Group 3. In this experimental group, the two environments are designed with various combinations of CPU and memory resources. Even though the behavior of each of the experimental applications is regular, it is very difficult to predict the optimum workload distribution in this type of environment. The results in Fig. 5 reveal that the performances of the SOR and Jacobi applications with the MEM-based policy are better than those achieved with the CPU-based policy in both test environments. This observation suggests that the behaviors of the SOR and Jacobi applications are more sensitive to memory resources than to CPU resources. As in Experimental Groups 1 and 2, Fig. 5 confirms that no matter how chaotic the combinations of CPU and memory resources, the proposed CPU& MEM-based policy consistently optimize the performance of all the experimental applications.

6. CONCLUSIONS AND FUTURE WORK

This paper has demonstrated the importance of incorporating memory resource considerations into the workload distribution scheme when attempting to improve the performance of DSM applications. The influence of memory resources on the DSM program performance has been thoroughly analyzed, and the results of this analysis have been applied to develop a novel workload distribution scheme implemented on a testbed referred to as Teamster. The proposed workload distribution scheme efficiently maintains the workload balancing of user DSM applications with comparatively low overheads. The experimental results have confirmed that memory swapping latency costs play a crucial role in determining the overall performance of DSM applications. The proposed workload distribution scheme outperforms conventional methods which consider the processor power only or the available physical memory only when attempting to minimize the execution time. The workload distribution scheme proposed in this paper is intended for clusters of computers having only one processor. However, an increasing number of computers have more than one processor nowadays. Accordingly, in a future study, the current study group intends to develop an advanced workload distribution method for DSM systems clustered with SMP (Symmetric Multiple Processors) machines.

REFERENCES

1. K. Thitikamol and P. Keleher, "Thread migration and load balancing in non-dedicated environments," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, 2000, pp. 583-588.
2. A. Dubrovski, R. Friedman, and A. Schuster, "Load balancing in distributed shared memory systems," *International Journal of Applied Software Technology*, Vol. 3, 1998, pp. 167-202.
3. C. Lai, C. K. Shieh, J. C. Ueng, Y. T. Kok, and L. Y. Kung, "Load balancing in distributed shared memory system," in *Proceedings of the IEEE International Performance*, *Computing*, *and Communications Conference*, 1997, pp. 152-158.
4. J. K. Hollingsworth and P. J. Keleher, "Prediction and adaptation in active harmony," in *Proceedings of the 7th International Symposium on High Performance Distributed Computing*, 1998, pp. 180-188.
5. K. Thitikamol and P. J. Keleher, "Thread migration and communication minimization in DSM systems," *IEEE Proceedings*, 1999, pp. 487-497.
6. W. Shi and Z. Tang, "Dynamic computation scheduling for load balancing in home-based software DSMs," in *Proceedings of the International Symposium on Parallel Architectures*, *Algorithms and Networks*, IEEE Computer Press, 1999, pp. 248-255.
7. T. Y. Liang, C. K. Shieh, and D. C. Liu, "Scheduling loop applications in software distributed shared memory systems," *IEICE Transactions on Information and Systems*, Vol. E83-D, 2000, pp. 1721-1730.
8. V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, "Distributed filaments: efficient fine-grain parallelism on a cluster of workstations," in *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994, pp. 201-212.
9. J. B. Chang and C. K. Shieh, "Teamster: a transparent distributed shared memory for cluster symmetric multiprocessors," in *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 508-513.
10. K. Thitikamol and P. J. Keleher, "Active tracking correlation," in *Proceedings of the*

19th International Conference on Distributed Computing Systems, 1999, pp. 324-331.

11. T. Y. Liang, J. C. Ueng, C. K. Shieh, D. Y. Zhuang, and J. Q. Lee, "Distinguishing sharing types to minimize communication in software distributed shared memory systems," *Journal of Systems and Software*, Vol. 55, 2000, pp. 73-85.
12. L. Xiao, S. Chen, and X. Zhang, "Dynamic cluster resource allocations for jobs with known and unknown memory demands," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, 2002, pp. 223-240.
13. J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Components*, Sun Microsystems Press, 2001, ISBN: 0-13-022496-0.

Yen-Tso Liu (劉晏佐) is currently a Ph.D. candidate studying at the Electrical Engineering Department of National Cheng Kung University, Tainan, Taiwan. He received his B.S. and M.S. degree from the Electrical Engineering Department of National Cheng Kung University in 1996 and 1998 respectively. His current research interests include distributed/parallel computing, parallel file systems, and computer networking.

Tyng-Yeu Liang (梁廷宇) obtained his M.S. and Ph.D. degrees from the Electrical Engineering Department of National Cheng Kung University in 1994 and 2000, respectively. Currently, he is an assistant professor studying and teaching at the Department of Electrical Engineering, National Kaohsiung University of Applied Science, Taiwan. His research interests include cluster and grid computing, and image processing.

Jyh-Biau Chang (張志標) is currently an associated professor in the Department of Information Management at Leader University in Taiwan. He received his B.S., M.S., and Ph.D. degrees from National Cheng Kung University in 1994, 1996, and 2005 individually. His research interest is parallel processing, distributed system, cluster and grid computing, and symmetric multiprocessing.

Ce-Kuen Shieh (謝錫堃) is currently a professor in the Department of Electrical Engineering, National Cheng Kung University. He received his B.S., M.S., and Ph.D. degrees from the Electrical Engineering Department of National Cheng Kung University, Tainan, Taiwan. His research interests include distributed and parallel processing systems, computer networking, and operating systems.