Multithreaded Self-Scheduling: Application of Multithreading on Loop Scheduling for Distributed Shared Memory Multiprocessor

K P Hung, N H C Yung and Y S Cheung
Department of Electrical & Electronic Engineering
The University of Hong Kong
Haking Wong Building, Pokfulam Road, HONG KONG

Abstract

A new loop scheduling scheme called multithreaded self-scheduling (MSS) for distributed shared memory multiprocessor is proposed. Based on the principles of multithreading, MSS attempts to hide the remote memory access latencies by switching between multiple contexts of threads. Consequently, loops scheduled by using MSS can obtain better performance comparing to the single-thread approaches.

In this paper, a series of simulation results corresponding to various parameter changes are presented, which provides a measure of the effectiveness of MSS under different boundary conditions and suggests the ways for further improvements.

1. Introduction

1.1 Distributed Shared Memory[1] [Li]

Shared memory multiprocessor offers a favourable programming paradigm of a global address space for parallel programs such that concurrent executing program components can communicate through shared variables. However, building an efficient network connecting all the processing elements can be expensive. Furthermore, such system has poor scalability.

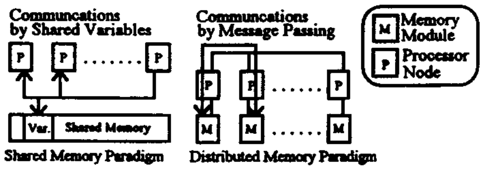

Communcations by Shared Variables
Communcations by Message Passing
Memory Module
Processor Node
Var.
Shared Memory
Shared Memory Paradigm
Distributed Memory Paradigm

Figure 1. Communications between Processor Nodes in Different Programming Paradigms.

On the other hand, distributed memory multiprocessor is more scalable, but its programming is clumsy and difficult as communications between processor nodes need to be explicitly coded using message passing *(Figure 1)*. Therefore, the concept of distributed shared memory multiprocessor is developed to take advantage of both systems' desirable characteristics and eliminate some of their pitfalls.

The hardware architecture of DSM multiprocessor is the same or very similar to that of distributed memory multiprocessor with the addition of a software or hardware layer *(a DSM abstraction)* to enable the memory modules distributed over processor nodes to form a global address space *(Figure 2)*. One of the characteristics of DSM multiprocessor is non-uniform memory access *(NUMA)*. When a memory access is issued, it will be faster if the information requested is located at the memory module of the local processor node. However, it will take longer if the information requested is located at the memory module of a remote processor node; though the mechanism for ensuring data consistency over the global address space, and the data transmission from remote memory module to local memory module are transparent to application program.

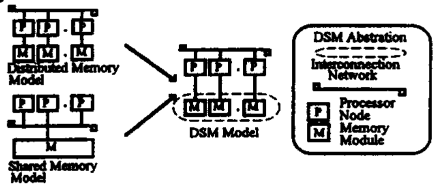


Figure 2. Different Memory Models for Multiprocessor

Although the abstraction of DSM can be implemented at different levels inside the computer hardware or system software, application software may not observe their difference in term of functionality *(Figure 3)*. For examples, Kendall Square Research's

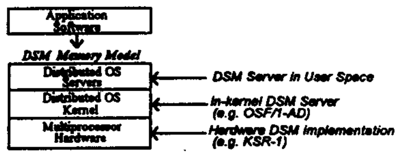


Figure 3. Implementation of DSM in different levels.

KSR-1[2] uses a hardware implementation of DSM called ALLCACHE[3], Open Software Fundation's OSF/1-AD[4] operating system supports an in-kernel DSM server. There are also many other institutes design user level DSM servers on top of common distributed operating systems such as Mach.

1.2 Latency Hiding Techniques

While DSM has merits of shared memory programming paradigm and more scalable, its memory latency problem demands serious consideration. Un-

[1]DSM is also known as Shared Virtual Memory (SVM)

[2]KSR-1 is a trademark of Kendall Square Research
[3]ALLCACHE is a patented invention of Kendall Square Research
[4]OSF/1-AD is a trademark of Open Software Foundation, Inc.

0-7803-2018-2/95/$4.00 © 1995 IEEE

like parallel program executing on distributed memory multiprocessor which may be scheduled and partitioned to match the underlying architecture for minimizing time consumption in remote memory accesses, DSM assumes a shared memory such that application programs running on top of it have no knowledge about local and remote memory accesses. Consequently, its number of remote memory accesses may be more than expected. Several latency hiding techniques [Hw] listed below have been developed to cater for this situation.

- prefetching
- coherent caches
- relaxed memory consistency
- multiple-contexts

Prefetching hides the latency of memory accesses by issuing them in advance and expecting them to be available when the executing program needs them [Sa]. Coherent caches try to reduce cache misses by hardware, and hence less remote memory accesses frequency resulted. Technique of relaxed memory consistency models is by pipelining and buffering memory accesses to hide the latency. Multiple-contexts technique attempts to hide latency by switching between contexts of different program execution components when a latency *(remote memory access or synchronization)* is encountered.

1.3 Processes versus Threads

To justify the technique of latency hiding using multiple-contexts, context switching time is a determining factor. The context of a process can be divided into two parts: system resources and execution states. Typical system resources associated with a process are addressable memory space, opened files, allocated communication ports, access control information, etc. They are often the large part of a process context especially the memory address space which contains a large buffer called table lookahead/lookaside buffer *(TLB)* for address space mapping. On the other hand, the context of execution state consists mainly of the processor registers, stack pointer, and program counter. It is often a small part of a process context.

In traditional operating system, a process *(Figure 4a)* supports only a single flow of execution *(also called thread).* Therefore, switching between different flow of executions requires to save and restore the corresponding process contexts which may take a very long time because of the significant system resources involved. In modern operating system, a process *(Figure 4b)* often supports multiple flow of executions *(multithread)*, and switching between different threads may be faster as only the contexts of execution states need to be saved or restored. For this reason, threads are often called light weight processes and the programs that contain multiple threads are often described as multithreaded programs.

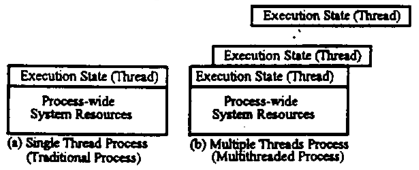

Execution State (Thread)
Execution State (Thread)
Execution State (Thread)
Process-wide System Resources
(a) Single Thread Process (Traditional Process)
Execution State (Thread)
Process-wide System Resources
(b) Multiple Threads Process (Multithreaded Process)

Figure 4. Traditional Process and Multithreaded Process.

1.4 New Loop Scheduling Scheme for DSM

In this paper, a new loop scheduling technique for DSM multiprocessor is proposed. By multithreading the chunks in guided self-scheduling *(GSS)* scheme, the remote memory access latencies, that frequently happen in DSM multiprocessor, may be effectively hidden by switching between multiple contexts of threads. Therefore, this new scheme is named as multithreaded self scheduling *(MSS).*

In order to compare and analyze the effectiveness in latency hiding by MSS, a series of simulation experiments were performed in comparing with the GSS. The simulation results suggest the boundary conditions for which MSS can obtain the best performance which may be useful as the criteria for improving both of the working mechanism of threads and the algorithmic approach of MSS.

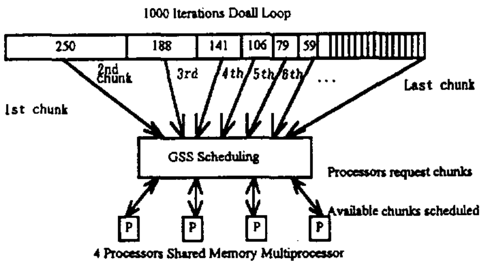
1.5 Organization of the Paper

This paper is organized into the following sections. Section 2 revisits some well known loop scheduling schemes for shared memory multiprocessor. Then, multithreaded self-scheduling scheme is introduced in Section 3 with an explanation of its working principles and its suitability for DSM multiprocessor. In section 4, a simulation model is developed, and some simulation cases are studied in section 5 so as to compare characteristics of multithreaded self-scheduling and guided self-scheduling schemes under different simulation conditions. Lastly, discussions on the simulation results and their implications are presented in section 6, and followed by a conclusion in section 7.

2 Loop Scheduling Schemes

Parallel loops are recognized as a great source of parallelism when parallelizing a program. Thus, a number of loop scheduling schemes [Lj] have been suggested. For a loop with no dependency between iterations, every iteration of the loop may be executed in parallel and it is sometimes called a doall loop. Doall loops can be scheduled on a shared memory multiprocessor statically *(prescheduling)* or dynamically *(self-scheduling).* Prescheduling assigns loop iterations evenly distributed on processor nodes. It expects no runtime overhead in scheduling and a good load balancing if the execution time for each iteration is the same. However, varying completion time for different iterations may result in imbalanced loading, and dynamic loop scheduling schemes are developed to solve this problem by moving the scheduling decision from compile or load time to run-time. The self-scheduling schemes allow processors responsible to allocate its own job. Scheduling one loop iteration at a time may introduce a significant scheduling overhead, and chunk scheduling scheme is proposed to schedule equal- size chunks of a number of iterations at a time to reduce this overhead. To further improve the load balancing performance of chunk scheduling, Guided self scheduling *(GSS)* [Po] is developed. It is a practical loop scheduling scheme which compromises between load balancing and scheduling overhead. GSS uses the strategy of decreasing chunk size for scheduling each successive chunk. The idea of GSS is to reduce scheduling overhead by scheduling chunks of coarser grain at the beginning, as well as to maintain a good balance of load by scheduling chunks of finer grain near the end. Referring to an example illustrated in Figure 5, a doall loop of 1000 iterations is scheduled on a four-processor shared memory multiprocessor by GSS scheme. For each job requested by a processor after completing a previous job, a chunk with iteration-size of c will then be scheduled, and c can be calculated by the following equation.

$$c = \lceil R/p \rceil$$, where R is number of iterations remained, and p is number of processors


1000 Iterations Doall Loop
250
188
141
106
79
59
2nd Chunk
3rd
4th
5th
8th
...
Last chunk
1st chunk
GSS Scheduling
Processors request chunks
Available chunks scheduled
P
P
P
P
4 Processors Shared Memory Multiprocessor
Figure 5. Example of GSS with Loop Count 1000 and 4 Processors

There are also some other scheduling schemes which derived from the GSS by further improving the load balancing *(Factoring)* [Hu] or scheduling overhead *(Trapezoid self scheduling)* [Tz]. One of the common features between all of these dynamic loop scheduling schemes is that they schedule a lot of loop iterations in a chunk at a time. This common feature does not only reduce the scheduling overhead but also allow multiple-contexts *(by multithreading)* latency hiding technique to be applied.

3. Multithreaded Self-Scheduling

Although the above self-scheduling schemes are well known as appropriate methods to schedule loops on shared memory multiprocessor, using the same technique on DSM multiprocessor may result in substantial performance degradation due to the large number of remote memory accesses. Consequently, multithreaded self-scheduling *(MSS)* scheme is proposed in this paper to address this issue.

In executing doall loops with large number of iterations on multiprocessor system, they are often divided into chunks. Each chunk contains a number of iterations and executes on an allocated processor node as a process. For example using GSS scheme in figure 6, a doall loop with loop count of 1000 may be divided into a number of chunks. These chunks can then be scheduled on processor nodes as smaller processes *(sub-tasks)*.

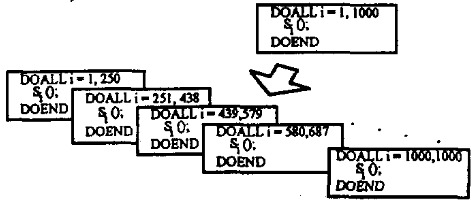
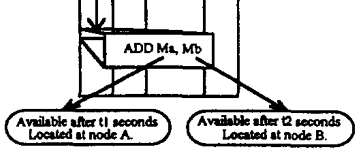


Figure 6. A Doall Loop divided into Chunks on a 4-processor Multiprocessor

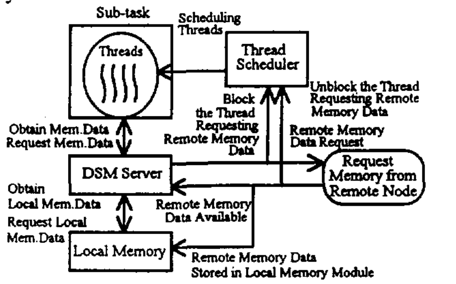
A sub-task can further be divided into smaller processes such that each iteration is itself a process. Furthermore, these one-iteration processes may share the same system resources in execution. Hence, it is appropriate to define them as threads and to encapsulate them into a process sharing the same system resources instead. This configuration of multithreaded sub-task supports multiple contexts of threads with efficient thread management operations. It is also the basic chunk defined in the scheme of MSS.

When a sub-task is executing on a processor node, situations may arise that latency is introduced. Two kind of latencies are often common in DSM multiprocessor, namely remote memory access latency and synchronization latency. Executing an instruction may involve some operands, and these operands may be located at different processor nodes. For example, as described in figure 7, the operands, Ma and Mb, are located at processor node A and B respectively. If the processor node, say A, is not the same node where the instruction is executing, this operand needs to be requested from the remote processor node A. Thus, a remote memory access latency would be expected. Moreover, the two operands, Ma and Mb, are likely to be available at different time t1 and t2 respectively. For this reason, the execution may have to wait until both of them are available. Thus, a synchronization latency *(t2 - t1)* would be expected.


S1
S2
.....
Sn
ADD Ma, Mb
Available after t1 seconds Located at node A.
Available after t2 seconds Located at node B.
Figure 7. A Typical Multithreaded Sub-task

Remote memory access latency may be considered as substantial. However, synchronization latency is difficult to forecast, it may be small or large and varies according to different program behaviours and execution environments. In MSS, only the remote memory access latency is intended to be hidden.

The working principle of MSS *(Figure 8)* is based on the cooperative work of DSM server and sub-task's thread scheduler. When a memory access is issued, the DSM server determines the availability of the information requested in the local memory module. If it is in the local memory module *(a hit)*, a local memory access is performed and the current executing thread continues. However, if it is not in the local memory module *(a miss)*, the DSM server resolves the memory access by requesting it from a remote processor node. There are various methods to perform this resolution in different DSM schemes [Ni]. In MSS, DSM server needs to acknowledge the thread scheduler on a miss, and the scheduler can base on this information to block the current executing thread and allocate the processor to another runnable thread. As the remote access is completed and the information is transferred from the remote memory module to the local memory module, the thread scheduler is acknowledged again such that the previously blocked thread can then be changed to a runnable thread for reallocation. If the time cost for managing the threads is small *(or cheap)* and there are sufficient number of threads for switching, the remote memory access latency may be effectively hidden.



Sub-task
Scheduling Threads
Threads
Thread Scheduler
Block the Thread Requesting Remote Memory Data
Unblock the Thread Requesting Remote Memory Data
Obtain Mem.Data
Request Mem.Data
Remote Memory Data Request
Request Memory from Remote Node
DSM Server
Obtain Local Mem.Data
Request Local Mem.Data
Remote Memory Data Available
Local Memory
Remote Memory Data Stored in Local Memory Module

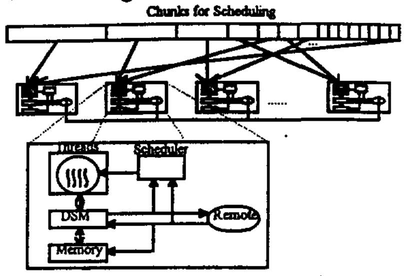
Figure 8. Block Diagram of Multithreaded Self-Scheduling

Although only GSS is multithreaded for our simulation study, MSS is a general technique which may be applied to most self-scheduling schemes with reasonable chunk size. Chunk self-scheduling, factoring, and trapezoid self scheduling are all possible be multithreaded. Furthermore, prescheduling can also be multithreaded as long as the chunk size is large enough.

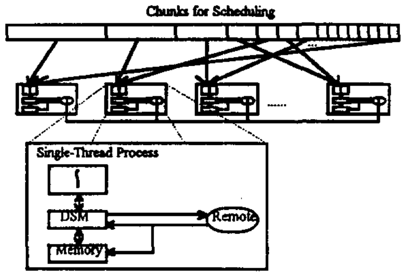
4 Simulation model

In order to compare the performance of the multithreaded self-scheduling scheme against the traditional self-scheduling scheme, a simulation model was built and tested based on both the MSS and GSS. The corresponding chunks scheduled in MSS are the same size to those of GSS. The difference is only on the behaviour of the chunks. In MSS, each chunk is a multithreaded process, while it is a single-thread process in GSS as depicted in figure 9.

The scheduler in figure 9a refers to the thread scheduler of MSS. It is responsible for multiple threads execution management. A thread can be of different states *(Figure 10)* and any change of its state is performed through the thread scheduler.



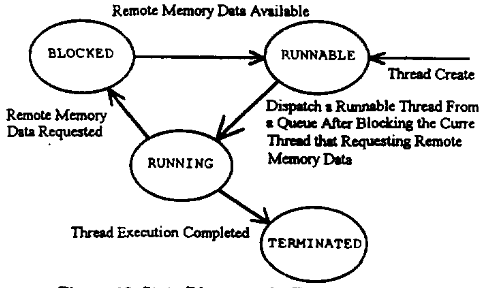
(a) Doall Loop Executed by Multithreaded Self-Scheduling



(b) Doall Loop Executed by Guided Self-Scheduling

Figure 9. Different Behaviour of Chunks in MSS and GSS.

For the sake of simplicity in the simulation, the overheads on creating and destroying threads are ignored by the assumption that the number of context switching operations between threads is sufficiently large compared with the thread creation and destruction operations, hence the total time of context switching operations dominates the overall execution time of a chunk contributed by thread management. This is often the case because thread creation and destruction are mostly the allocation and deallocation of small storage for the execution states (processor registers, stack pointer, program counter). These overheads are generally small though it depends on the specific system and thread implementation method. In addition, the number of context switchings is substantial in MSS as remote memory accesses are common in DSM multiprocessor, and each remote memory access triggers at least one context switching *(one for*



Remote Memory Data Available
BLOCKED
RUNNABLE
Thread Create
Remote Memory Data Requested
Dispatch a Runnable Thread From a Queue After Blocking the Curre Thread that Requesting Remote Memory Data
RUNNING
Thread Execution Completed
TERMINATED

Figure 10. State Diagram of a Thread in MSS.

blocking this thread, and maybe another one for dispatching this thread later when the remote memory *data is available).* Furthermore, synchronizations between threads are also ignored by the fact that there is no dependency between threads in doall loops.

Intuitively, the trade-off between MSS and GSS is the context switching time and the remote memory access latency. Therefore, several simulation cases were investigated and simulated with varying simulation parameters in order to study this intuition in details.

5. Simulation Cases

The doall loop used in the simulation is shown in figure 11. It contains no inter-dependency between different iterations within the loop, and each iteration is characterized by the portion of ExeOnly *(execution time units without memory access)* and the portion of ExeMem *(execution time units with memory access)*.

```
DOALL I = 1, N
    ith_iteration ( ExeOnly,  ExeMem )
ENDDO
```

Figure 11. A Doall loop for the simulation experiments.

Throughout the simulation, the distribution of two kinds of execution time units *(ExeOnly and ExeMem)* in an iteration is assumed to be random because memory access distribution in a set of instructions is determined by the specific application, the coding method of the programmer, as well as the code generation method of the compiler. With such complex factors affecting the memory access pattern, it is almost impossible to forecast when a memory access will be issued. Therefore, the use of random memory access pattern seems appropriate in this simulation study. Moreover, the occurrence of local memory access is also assumed to be random and fixed by a hit ratio. The reason for assuming random occurrence of local memory access is very similar to that of memory access pattern because it is also affected by programming method, code generation method, specific application as well as DSM schemes. The latencies of different kinds of memory access are fixed to a constant so that remote memory access latency is a fixed time period everytime and so as the local memory access latency. It is the simplest model of a NUMA multiprocessor without considering the variation of the memory access latency. In real situation, it is often acceptable as the time difference between local memory access and remote memory access is large so that the latency variations of these two kinds of memory accesses are very localized. Therefore, an acceptable approximation of them are using constant values as in this simulation.

Context Switching Time:	**200 or 400 units**
Local Memory Accesses Hit Ratio:	**97 %**
Local Memory Accesses Latency:	**10 units**
Remote Memory Accesses Latency:	**1000 units**
ExeOnly (Per Iteration Without Memory Access Execution Time Units):	**1000**
ExeMem (Per Iteration With Memory Access Execution Time Units):	**500**
Number of Loop Iterations:	**1000**
Number of Processors:	**20**

Figure 12. Default Simulation Parameters.

Two sets of simulation experiments are performed and they are based on different context switching times, namely 200 and 400 time units. Unless a specific simulation parameter is being varied, the default values for these parameters are given in figure 12.

These parameters are carefully chosen with the objective of reflecting some realistic situations.

- The default context switching time is 200 or 400 units which may reasonably reflect the context switching time for saving and restoring processor contexts. We do not assume a fast context switching time because most of the thread implementations currently are performed at the software level. User-space threads have faster context switching time, while kernel-level threads have slower context switching time.
- The default hit ratio is 97% because the effect of locality is assumed; though it may vary on swap page size, specific application, etc.
- The default local memory access latency is 10 units and that of remote access latency is 1000 units. The local memory access time is very fast by its nature and is likely to be much faster than the context switching time, therefore 10 units are assumed. On the other hand, the remote memory access is generally considered as a slow process affected by the speed of the interconnection network and its contentions, routing algorithm, and DSM server overhead. Consequently, it is assumed to be several times slower than the default context switching time.
- *ExeOnly and ExeMem* are application specific and a reasonably long iteration is assumed.
- The default number of loop iterations is 1000 while the default number of processors is 20 so that a reasonably large chunk size for self-scheduling schemes is resulted. The numbers are not chosen to be excessively large so that the simulation cases can be completed within a reasonable time period.

$$\textbf{Efficiency} = \frac{\textbf{Average Processor Busy Time}}{\textbf{Overall Execution Time}},$$

where processor is busy means that the processor is producing output

One of the aims for applying latency hiding techniques in DSM multiprocessor is to improve the efficiency of the processors by minimizing their idling time and keeping them busy as often as possible. Therefore, we present also the simulation results in the characteristics of processor efficiency by the following formula.

5.1 Effect of Varying Thread Context Switching Time

This simulation experiment is performed by varying the context switching time from 10 to 1500 time units. MSS's overall execution time decreases approximately linearly as the context switching time decreases. However, GSS shows a behaviour that is insensitive to context switching time changes. The two graphs intercept at a point with context switching time of 970 time units and execution time of 910,000 time units. For context switching time smaller than 970 time units, MSS performs better than GSS and reaches 35,000 execution time units as context switching time approaches to zero. Beyond 970 time units of context switching, MSS performs poorer than GSS.

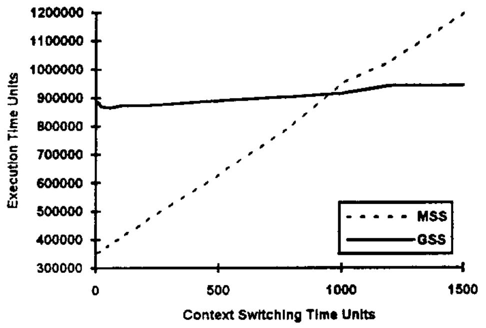


Figure 13. Execution Time vs Context Switching Time

Similar to the case of execution time versus context switching time, two graphs of processor efficiency versus context switching time intercept at a point with context switching time of 970 time units. However, MSS shows an exponential increase of processor efficiency as context switching time decreases. When the context switching time approaches to zero, the processor efficiency approaches 21.5% which is more than 2 folds better than that of GSS's 8.5%.

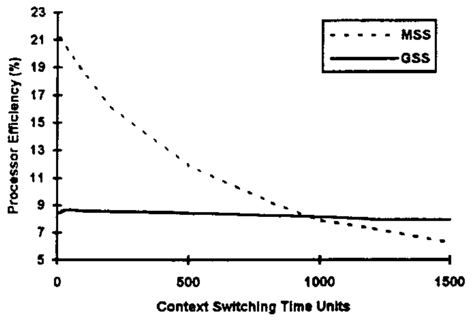


Figure 14. Processor Efficiency vs Context Switching Time

5.2 Effect of Varying Local Memory Accesses Hit Ratio

This simulation experiment is performed by varying the local memory accesses hit ratio from 70% to 99%. For comparing the overall execution time as in figure 15, MSS performs better than GSS in this range of hit ratios. Furthermore, it may be expected by projection to have a higher improvement if the hit ratio goes below 70%. MSS has relatively small variation in overall execution time in this range *(e.g. from 360,000 to 1894,0000 in MSS-200)* compared with that of GSS *(from 360,000 to 7680,000 in GSS-200)*. However, all the graphs converge as the hit ratio approaches to 100%. To an extreme, hit ratio of 100% is actually the special case of shared memory multiprocessor.

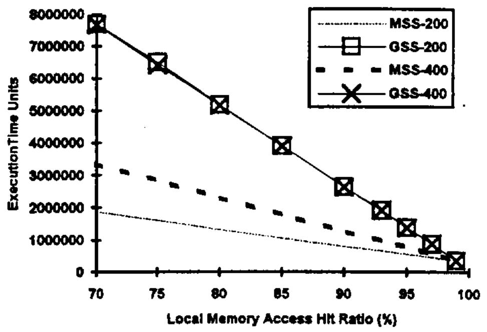


Figure 15. Execution Time versus Local Memory Accesses Hit Ratio

While execution time shows a linear relationship with the local memory access hit ratio, the processor efficiency exponentially improves as the hit ratio increases. With very high hit ratio, say 99%, both MSS and GSS have processor efficiency of 21%. However, GSS shows a substantial decrement *(8% at 97% hit ratio, 3% at 90% hit ratio for GSS-200)* as the hit ratio decreases. Relatively speaking, MSS suffers smaller processor efficiency degradation on decrement of hit ratio *(17% at 97% hit ratio, 9% at 90% hit ratio for MSS-200)*.

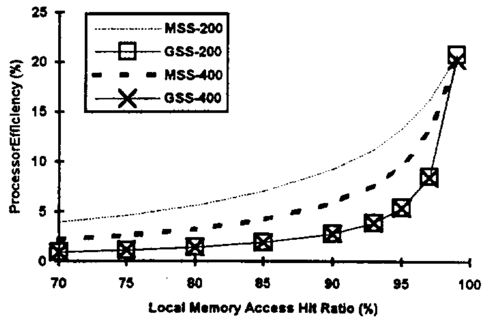


Figure 16. Processor Efficiency versus Local Memory Accesses Hit Ratio

5.3 Effect of Varying Local Memory Access Latency

This simulation experiment is performed by varying the local memory access latency from 10 to 250 time units. All the graphs in the plot of overall execution time versus local memory access latency are linear and very close to each other. However, MSS shows a constant improvement in overall execution time compared with the GSS in this range of local memory access latency.

In processor efficiency, the graphs decrease expo-

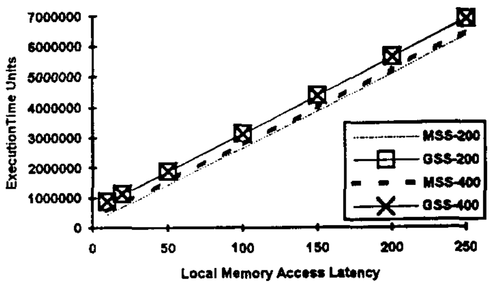


Figure 17. Execution Time versus Local Memory Access Latency

nentially as the local memory access latency increases. For the latency larger than 100 time units, all 4 graphs are sufficiently close to each others such that no significant difference in processor efficiency can be found. However, for the latency smaller than 100 time units, MSS *(16% at 10 time units latency)* shows a two-fold improvement compared with GSS *(8% at 10 time units latency).*

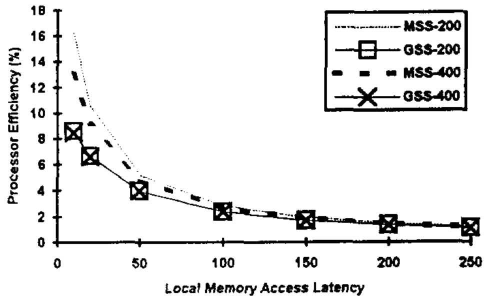


Figure 18. Processor Efficiency versus Local Memory Access Latency

5.4 Effect of Varying ExeMem (Per Iteration With Memory Access Execution Time Units)

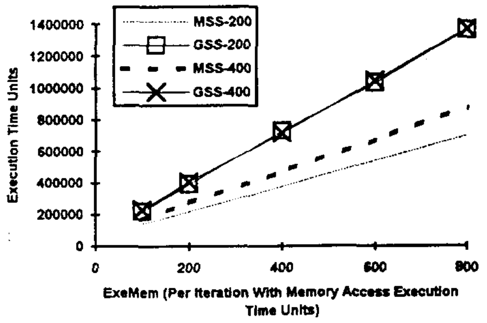


Figure 19. Execution Time Units versus ExeMem

This simulation experiment is performed by varying the ExeMem from 100 time units to 800 time units. Figure 19 shows that the execution time performance of MSS is always better than GSS in this range of ExeMem. As the ExeMem increases, improvement in overall execution time for MSS from GSS increases. All the graphs show a linear relationship between overall execution time and ExeMem, and are expected *(by projection)* to converge at a point near zero ExeMem.

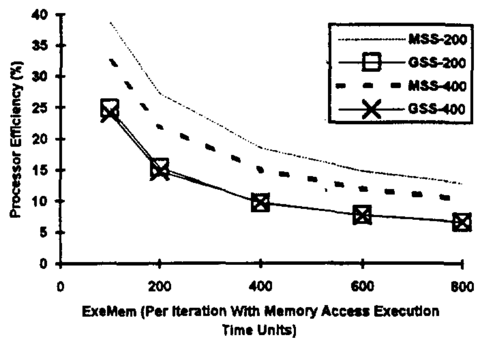


Figure 20. Processor Efficiency versus ExeMem

Processor efficiency increases exponentially as the ExeMem decreases. In this range of ExeMem, MSS always performs better than GSS *(38% at 100 ExeMem for MSS-200 versus 24 % at 100 ExeMem for GSS-200 and 13% at 800 ExeMem for MSS-200 versus 7% at 800 ExeMem for GSS-200).*

5.5 Effect of Varying Number of Processors in the Multiprocessor

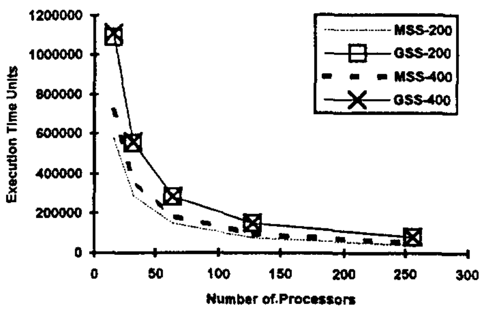


Figure 21. Execution Time Units versus Number of Processors

This simulation experiment is performed by varying the number of processors from 16 to 256. The overall execution time increases exponentially as number of processors decreases. The graphs almost converge at the point with 256 processors while MSS shows substantial improvement over GSS with smaller number of processors *(e.g. 1100,000 time units at 16 processors for GSS-200, and 570,000 time units at 16 processors for MSS-200).*

Processor efficiency decreases linearly as the number of processors increases. However, the graphs show a small variations in the processor efficiency changes with variation in the number of processors

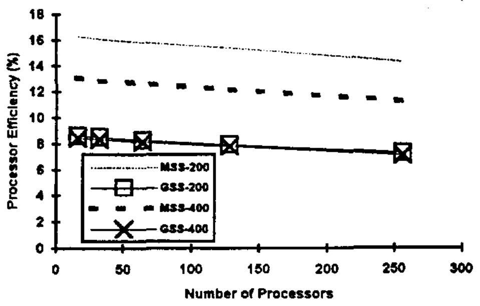


Figure 22. Processor Efficiency versus Number of Processors

(16.2% at 16 processors and 14.2% at 256 processors for MSS-200, while 8.2% at 16 processors and 7.2% at 256 processors for GSS-200).

6. Discussions

6.1 Explanation of the Simulation Results

When MSS and GSS are tested using different values of context switching time, we can observe in figures 13 and 14 that MSS shows a substantial improvement over GSS as the context switching time decreases. In fact, GSS is rather insensitive to the change of context switching time. This is also shown in the results of other simulation experiments in which the behaviours of GSS are very similar using different context switching time of 200 or 400 units. It can be explained as GSS does not switch between iterations, and therefore the context switching time only contributes a small percentage to the overall execution time.

Ratio of MSS to GSS Execution Time (%)	Ratio of Context Switching Time to Remote Memory Access Latency (%)
39.0	1.0
46.5	10.0
70.8	50.0
88.9	80.0
100.0	97.0
103.8	100.0
126.8	150.0

Table 1. Ratio of MSS to GSS Execution Time with different Context Switching Time.

The intercepting point of the two graphs occurs when the context switching time is slightly less than the remote memory access time as referred to table 1. At the point when context switching time is the same as that of the remote memory access latency, MSS is slightly slower than GSS. This is due to the non-productive context switchings. To hide the memory latency, multiple-contexts are switching on encountering remote memory accesses. When there are a lot of runnable threads, blocking a thread by switching to a runnable thread may effectively hide the latency. However, when runnable threads are exhausted, blocking a thread by switching may be redundant because there is no other runnable thread that can use the processor. It may prolong the latency by switching back when it becomes a runnable thread again upon the availability of the remote memory data.

Beyond the intercepting point, overhead of context switching becomes dominant, as such MSS results in a poorer performance compared with the GSS. This simulation result suggests that MSS may be an effective way for latency hiding only if the context switching time is reasonably smaller than the remote memory access latency. Furthermore, the per processor efficiency is improved substantially when context switching time is smaller than 30% of the remote memory access latency as indicated in figure 14. The processor efficiency is improved approximately 2% per each 100 time units of context switching time decreases within the range of 0 to 300 time units of context switching time, while it is about 1.14% in the range of 300 to 1000 time units and even smaller for the range larger than 1000 time units. However, to shorten the context switching time in the former range may require substantially more efforts compared with the later range.

The second simulation experiment investigates the effect of varying the local memory access hit ratio to the loop scheduling methods. From figures 15 and 16, we can observe that both MSS and GSS converge to one point as the hit ratio approaches to 100% which is also the special case of shared memory multiprocessor. It is because the number of remote memory accesses approaches to zero and hence the thread switching benefits no longer exist. As the hit ratio decreases, the performance improvement of MSS is more significant. From this result, it is suggested that GSS and MSS have very similar performance on shared memory multiprocessor, while MSS may be more suitable for DSM multiprocessor if the simulation parameters hold.

Hit Ratio (%)	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 200)	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 400)
70	2.99	1.27
80	4.18	1.80
90	6.42	3.06
93	7.35	3.76
95	7.93	4.27
97	7.67	4.61
99	0	0

Table 2. Percentage Improvement on Processor Efficiency with different Local Memory Access Hit Ratio.

Another interesting observation is that the percentage improvement of processor efficiency is not monotonous as depicted in figure 16. Moreover, some quantitative comparisons can be found in table 2. The percentage improvement reaches the highest point at hit ratio of around 96%. It may be explained as follows. For the case with very high hit ratio, the number of remote memory accesses is small such that the merit of MSS in latency hiding cannot be effectively shown. Hence, a relatively small processor efficiency improvement. On the other hand, a very low hit ratio results into a large number of remote memory accesses and relatively insufficient available threads for effective multiple-contexts latency hiding. Therefore, the processor efficiency improvement is relatively small too. For an optimal point, the hit ratio should not be too high or too low such that the number of threads matches the number of remote memory accesses for the best latency hiding effect. In short, the number of iterations *(threads)* in a chunk and number of remote memory accesses may require to match in order to obtain the best processor efficiency improvement by MSS.

Referring to figure 17, we can observe that the variation of local memory access latency has no significant effect on the overall execution time difference between MSS and GSS. It can be interpreted as the improvement on latency hiding will not be affected by this parameter. It is logical as the switching of threads is merely determined by the signal from the DSM server to the thread scheduler which in turns is decided by the nature of the memory access *(local or remote)*. If the local memory access latency is large compared with the context switching time, the performance of MSS would be improved further by switching threads on encountering any kind of memory accesses.

Since the number of local memory accesses is large for the default hit ratio *(97%)*, the improvement on processor efficiency becomes insignificant when the local memory access latency increases as shown in figure 18. There is no reason to spinning *(spinning means the thread will continue to execute a tight loop in which it waits for the event)* on local memory access when the local memory access latency is comparable to that of the remote memory access.

ExeMem	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 200)	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 400)
100	13.8	8.4
200	11.8	7.3
400	8.9	5.1
600	7.0	4.3
800	6.2	3.6

Table 3. Percentage Improvement in Processor Efficiency with different ExeMem.

By varying the ExeMem *(per iteration with memory access execution time units)*, we can observe from figure 19 that MSS always performs better than GSS with proportional increment as ExeMem increases. However, the percentage improvement on processor efficiency is slightly deduced *(figure 20 and table 3)* as ExeMem increases. It can be explained that the fewer the memory accesses *(and hence fewer remote memory accesses)* the better the MSS can hide the remote memory access latency. It is expected that the improvement in processor efficiency may drop again when the number of memory accesses is very small which is similar to the case with high local memory access hit ratio.

The last simulation experiment is performed by varying the number of processor nodes in the multiprocessor. This in turns varies the average chunk size on scheduling. From figure 21, it is observed that the performance improvement of MSS is more significant when the number of processors is small *(execution time improved from 1100,000 to 570,000 time units at 16 processors)*. In addition, figure 22 and table 4 show that the percentage improvement on processor efficiency of MSS and GSS is converging slowly.

No. of Processors	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 200)	Percentage Improvement in Processor Efficiency from GSS to MSS (CS Time = 400)
16	7.7	4.7
32	7.6	4.5
64	7.6	4.5
128	7.5	4.4
256	7.0	4.2

Table 4. Percentage Improvement in Processor Efficiency with different Number of Processors.

For the case with fewer processors, the average chunk size is relatively large and multithreading on a larger chunk may result in a better latency hiding effect. However, as the number of processor nodes increases, the performance of both GSS and MSS converges. In one extreme, the chunk size reduces to one when the number of processor increases to 1000. In this situation, the MSS and GSS have no difference as processor nodes in both schemes have only one thread to execute. Of course, the task completion time is faster with an increasing number of processors but the processor efficiency is poorer. Actually, scheduling one iteration at a time is not a good idea as the scheduling overhead is great, a slight modification of GSS had also been suggested in [Po] to define somehow a minimum chunk size which may also be beneficial to MSS.

6.2 Different Level of Threads

In the above simulation study, we have not assumed any specific threads implementation as well as the details of the DSM. Since their implementation decisions may be closely related, threads *(like DSM)* can also be implemented at different levels. As we have discussed the implementation choices of DSM before, let us look at the issues on threads now. In practice, threads can be classified as user-level threads and kernel-level threads. User-level threads implementation can result in fast thread management operations because the thread scheduling is in the user space. However, the scheduler has no way to access the information in the kernel and hence only non-blocking type kernel system calls may be used which are usually slower. In contrary, kernel-level threads implementation does not have this problem as the scheduler can obtain the kernel information for thread management but it suffers a serious performance drawback as thread management needs to be performed through system calls. Furthermore, the crash of a kernel thread may corrupt the kernel so that protection checking on kernel thread operations has to be performed which is time consuming.

A hybrid kernel/user-level thread management system based on scheduler activations have been suggested which contains the most benefits of user-level thread and kernel-level thread [An] [Da]. The idea of this management system is that user-level threads are built on top of a kernel entity called scheduler activation. Scheduler activation supports communications between user-level threads and the kernel by notifying the user-level threads of kernel events and vice versa. Therefore, the performance of such threads is good for they are executing in user-level, as well as retaining the functionality of kernel level threads.

7. Conclusion

From the above performance analysis of MSS and GSS, we conclude that MSS is an efficient loop scheduling scheme for DSM multiprocessor. However, several considerations may be desirable in order to obtain the best performance from MSS.

- The context switching time between threads needs to be small compared with the remote memory access latency. For our default simulation parameters, the context switching time of 30% or below of *(i.e. below 300 time units)* the remote memory access latency can result in 1.8 to 2.6 times processor efficiency improvement or 1.7 to 2.4 times execution time improvement by using MSS instead of GSS. For the case with context switching time comparable to or larger than the remote access latency, MSS performs poorly.
- The number of remote memory accesses and the number of threads on a multithreaded sub-task need to be matched for the best processor efficiency. Therefore, calculation of chunk size *(number of threads)* can be modified from the method of GSS with consideration given to the number of remote memory accesses.
- Local memory accesses may also be hidden if context switching time is sufficiently small comparing to local memory access latency.
- MSS is more efficient in the situation where the number of processors in the multiprocessor is scarce and the loop *(or total number of threads)* is relatively large.

Several issues related to this simulation can be further investigated. A more sophisticated simulation model which considers both the memory locality effect of DSM and some real problem loops can be studied. A machine realization of MSS is now under investigation which would reflect the real execution environment more accurately. Further investigation on applying the concept of MSS to other scheduling schemes is in progress, which may eventually cover the doacross loop and with other dependencies.

8. References

[An]T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proc. of the 13th ACM Symposium on Operating System Principles, in Operating System Review, Vol 25, No. 5,* pp 95-109, Oct. 1991.

[Da]P. B. Davis, D. McNamee, R. Vaswani, E. D. Lazowska, "Adding Scheduler Activations to Mach 3.0," *Technical Report 92-08-03, University of Washington*, Aug. 1993.

[Hu]S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Proc. Supercomputing 91, IEEE CS Press,* pp. 610-619, 1991.

[Hw]K. Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability," *Computer Science Series, McGraw Hill, Inc.*, pp. 475-504, 1993.

[Li] K. Li, "Shared Virtual Memory System on Loosely Coupled Multiprocessor," *Technical Report, Yale University*, 1986.

[Lj] D. J. Lilja, "Exploiting the Parallelism Available in Loops," *IEEE Computer*, Feb. 1994.

[Ni]B. Nitzberg, V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer, Vol. 24*, pp. 52-60, Aug. 1991.

[Po]C. D. Polychronopoulos, D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers, Vol 36, No. 12*, pp 1425-1439, Dec. 1987.

[Sa]R. H. Saavedra, W. Mao, K. Hwang, "Performance and Optimization of Data Prefetching Strategies in Scalable Multiprocessors," *Journal of Parallel and Distributed Computing Vol. 22*, 1994.

[Tz]T. H. Tzen, L. M. Ni, "Trapezoid Self Scheduling: A Practical Scheduling Scheme for Parallel Computers," *IEEE Trans. Parallel and Distributed Systems, Vol. 4, No. 1*, pp. 87-98, Jan. 1993.