Contents

Scheduling Parallel Workloads : Implicit Scheduling vs Co-scheduling

Suman Banerjee *Gayathri Chittiappa*

May 20, 1997

Abstract

In this project we evaluate and compare the benefits of scheduling parallel jobs using different scheduling techniques. The traditional co-scheduling technique schedules all processes of the same job to be active simultaneously. However, implicit scheduling, where each processor makes independent decisions to schedule processes of different jobs might be competitive with co-scheduling under certain constraints and assumptions. In this project, we implemented a discrete event simulator to compare the performance of both these techniques and search for methods to gain the performance benefits of co-scheduling without the costly requirement of a centralized scheduler.

1 Introduction

The basic intuition behind co-scheduling [1], where processes of the same job are scheduled in the respective processors at the same time, is that communicating processes are ready to respond to messages, which thus ensure that messages are served as soon as they are received. This is an advantage in tightly coupled systems, where the message transmission latencies are low. As a consequence, not only the overhead of context switch is avoided, but also a process that is awaiting a message can be released quickly.

However, with the emerging distributed environments for processing parallel workloads, the above technique may not be best suited. The implicit scheduling [2] scheme uses a local scheduler to decide on which process to schedule and when. By using the priority assignment to processes to decide on which process to schedule, a process that is not CPU-intensive will get a better share of the processor as and when it requires.

This would also be very useful also in the situation when both interactive and I/O intensive jobs are mixed with batch jobs. To improve throughput, jobs waiting on I/O relinquish the processor, and are given a higher priority when they have computation. Also, since local scheduling exists on each machine, it requires no additional implementation.

Hence, the basic requirement to get good performance using local scheduling is if the process can decide properly, when to relinquish the processor to another process. If a good estimate of the I/O or message blocking can be estimated, a process can be optimally scheduled so that idle time is minimized thus improving the utilization of the processor.

The implicit scheduling technique as suggested [2], uses a two-phase adaptive blocking mechanism to decide on if and when to de-schedule an idle process. The process is allowed to spin wait for twice its context switch time, and if a message returns within that period of time, it would be more efficient to allow that process to continue than to context switch out. However, if no response is received in that time, a blocked process is scheduled instead of the current one.

2 Focus

The prior work in implicit had some specific assumptions about the workload modeling and the kind of environments in which the jobs were to be executed. They had assumed a bulk-synchronous SPMD model of parallel jobs, where phases of computations and communication were clearly separable. Also, they had used a fixed and arbitrary context switch time for all the processes, which did not properly reflect the actual overhead of the workloads being simulated.

In this project, we focussed on more generic applications, namely Barnes-Hut, Water and Sor, and compared the performance of the two techniques with various combinations of these jobs competing for the limited processing resource.

Also, we used a better estimate of the context switch time for each process to better decide on the period over which a process should spin wait. A more dynamic scheme of process context switch times which varies over the execution lifetimes of the processes might be more helpful in making the decisions.

3 Methodology

We used CVM [3], a software based distributed-shared memory system, to extract traces for the various applications. The traces were time-stamped sequence of events that were reported happen in each of the process of a job. To get the traces, we instrumented the CVM source code. We used the *gettimeofday*() routine to figure out instants at which the events occur. Hence, although we measure the timing information to the granularity of microseconds, the actual accuracy may be a magnitude of order larger. Then we implemented a discrete event simulator on which to re-run these traces, using both the co-scheduling and implicit scheduling techniques. The simulator was allowed to re-order the sequence of events from different processors, since only the necessary dependencies like lock requests can follow lock releases, need to be preserved and the actual ordering will depend on the exact run-time environment. We finally compared the performance statistics that we got from the simulator.

4 Parallel Workload Modeling

For our purposes, a parallel workload is expected to comprise of – Computation, Lock Requests, Lock Releases, Memory Requests and Barrier Synchronization.

Computation has associated with it a specific computation duration, during which the process does not communicate with any of its peers. Lock requests and releases are made for some specific locks – each with its unique lock identifier. The lock requests are modeled as a distributed queue, which is served in the FIFO order. Each lock has a initial lock manager, to which the first lock request is made. Each subsequent lock request is then made to the previous owner of the lock, who becomes the manager for that period. Each lock request is, thus, serviced by the use of only two messages – a lock request and a lock grant. Obviously, lock requests made by the process to itself (it the same process is the lock manager) is served without any exchange of messages. When a lock is released by a process, it sends a lock grant message to the requester if there are any waiters in the distributed queue.

Memory requests are can be made any process to any of its peers, and the memory request is served with a message, whose size is determined by the size of memory that is being returned. Barrier arrival messages are sent by all the processes to a pre-determined barrier master process. The barrier master sends out barrier release messages to all the other processes once it has got the arrival message from all the processes, including itself.

Any kind of request message sent results in the requesting process to block till the response for the process is received. A process services requests for memory, requests for locks and handles barrier arrival messages with a higher priority than its local computation. For the purpose of simplicity it has been assumed that all messages that are not memory request responses have a fixed size. Also, we assume that processing of messages have no overhead, except the time spent in receiving the same, which depends on the size of the message.

5 Simulator Description

The discrete event simulator that we implemented has various kinds of events emulating the actual execution of a parallel job. The events are – raw computation, message sends and receives for barrier arrivals and releases, lock requests, grants and releases and memory requests and replies, context switch in and out. Each event at its termination schedules more events that is expected to follow.

The simulator is designed as a collection of event queues, one for each process that is present in the system. There is also a future event schedule, which places events destined to happen in the future, into the process event queues. All the queues are driven by a timer, which de-queues the most recent event from the queue, takes appropriate actions for the event (i.e. schedules some other events) and updates the current time.

All messages are assumed to be atomic, i.e. they cannot be split across a context switch. For ease of implementation, we handle the errant cases by terminating the message at the beginning of context switch.

The scheduling module has been made separate from the design of the rest of the simulator so that we can easily replace one scheduling mechanism with another. We implemented the co-scheduling technique where context switches across all processes occur at the same time.

We also implemented priority based implicit scheduling, where the context switch decisions are made independently by each processor. The scheduling in each processor is priority based. We used the dynamic priority assignment scheme as used for 4.3 BSD Unix [4]. In this, the system clock ticks 100 times a second, and the priorities of the processes are updated on every 4^{th} tick. The CPU usage of a process is increased by one each time the process is found to be using the CPU. The CPU usage is decayed by half the current value once each second when it did not use the CPU. Here, processes are switched out when its time slice expires or it is still busy waiting with the CPU with its spin-wait phase over.

To contrast the utility of the priority based scheduler over a simple round-robin scheduler, we also implemented a round-robin local scheduling policy, where a new process for a given processor is picked from the ready to run queue in a round robin fashion. The decision to switch out a process is made as described for a priority based local scheduler.

6 Performance and Evaluation

To illustrate and compare the performance of the two scheduling techniques we varied the message latency and ran three and five jobs simultaneously on our simulator. The other parameters that we used in the simulation were as follows –

- Time slice 10 ms.
- The message sizes for various events (as observed from the CVM traces) were – Lock request 50 bytes, Lock release 120 bytes, Lock grant 80 bytes, Page request 24 bytes, Diff request 40 bytes, Barrier request 88 bytes and Page size 4096 bytes.
- We assumed that in case of context switch out for a process, the overhead is a constant, where

as for the context switch in the time required is dependent on the working set size of the process.
- The message latencies that we used were 50 μs, 300 μs, 3000 μs,10000 μs.
- The working set sizes are – Barnes (100 KB), Water (50 KB), Sor (800 KB). They were approximated by the per process shared memory sizes, which is given by actual shared memory size/ number of processors. The corresponding context switch times are – Barnes (200 μs), Water (100 μs) and Sor (1600 μs).
- Each job was alloted 4 processors for the experimentation.
- The three jobs – Barnes, Water and Sor have different characteristics. Sor is a computation intensive job, whereas Barnes and Water both do frequent communication, Barnes makes a lot of memory requests, while Water does more lock requests and barrier synchronization.
The following tables illustrate the performance characteristics of the different scheduling techniques (the percentage times for each process of a job are given split between the Idle, Busy, Blocked and Switching times).

Message latency 50 μs, Barnes+Water+Sor (3 jobs) % times

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit)	5.17	23.52	65.61	5.71
Barnes (Cosched)	4.52	26.26	67.99	1.23
Water (Implicit)	1.78	28.98	66.65	2.59
Water (Cosched)	0.90	30.01	68.17	0.93
Sor (Implicit)	4.89	38.68	51.50	4.93
Sor (Cosched)	5.37	41.73	48.64	4.26

For message latency of 50μs, the spin-wait phase (2 $\times$ context switch time) is more than the message latency. Here, the simulated underlying architecture is a set of tightly coupled processors. Hence, it is expected that the processes rarely spin wait (only in the case of lock requests when the lock is currently being held by another process). As a consequence, the the implicit scheduling case degenerates to pure local priority scheduling, with each process occupying the CPU for the entire time slice. In this situation, co-scheduling as expected, performs better than implicit scheduling. This can be noticed by observing that the busy time to idle time ratio is more in co-scheduling than in implicit scheduling.

Message latency 300 μs, Barnes+Water+Sor (3 jobs) % times

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit)	7.07	19.96	65.33	7.64
Barnes (Cosched)	6.01	24.76	67.99	1.23
Water (Implicit)	3.45	24.40	66.65	5.50
Water (Cosched)	3.63	27.51	67.93	0.93
Sor (Implicit)	4.52	34.65	55.41	5.42
Sor (Cosched)	3.36	39.79	50.38	4.46

When the message latency is increased to 300μs, the spin-wait phase for Sor (1600 μs) is still greater than the message latency but not so for Barnes (400μs) and Water (200μs). In this case, Sor still favors co-scheduling, whereas in Water, the advantage of co-scheduling has dwindled. This is because, in Water, processes are spin-waiting less in implicit scheduling than in co-scheduling. Barnes, however, has not shown much appreciable change from the previous case. In fact, it is worse off than the last case (message latency 50 μs), which could possibly be explained by noting that the Barnes spends a lot of time context switching. In fact, Barnes spends a lot of time spin-waiting, and is context switched out a little before the message returns, which is more wasteful.

The trend for increasing the message latencies is clear from the following two cases with even larger latencies.

Message latency 3000 μs, Barnes+Water+Sor (3 jobs) % times

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit)	9.99	14.09	65.53	10.40
Barnes (Cosched)	19.08	12.44	67.23	1.24
Water (Implicit)	6.21	17.47	66.65	9.67
Water (Cosched)	15.88	14.88	68.31	0.92
Sor (Implicit)	4.32	27.83	61.74	6.11
Sor (Cosched)	5.02	26.19	63.16	5.62

Message latency 10000 μs, Barnes+Water+Sor (3 jobs) % times

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit)	23.42	8.10	55.17	13.31
Barnes (Cosched)	37.30	6.30	56.25	0.15
Water (Implicit)	10.81	10.59	62.48	16.12
Water (Cosched)	32.35	7.71	59.81	0.12
Sor (Implicit)	4.79	25.95	66.17	3.10
Sor (Cosched)	8.89	24.16	66.35	0.59

In the above two cases, it is clear that implicit scheduling is a better option than co-scheduling. In both these cases, the spin-wait phase for processes are much smaller than the round trip time of messages. As a consequence, CPUs can be kept more busy if processes context switch out, instead of spin-waiting for message responses.

Although the process spin-wait times have decreased sharply, note that the time spent context switching has gone up correspondingly in implicit scheduling. This is expected because in implicit scheduling the processes context switch in and out more often than in co-scheduling for larger message latencies. For message latencies of 10000 μs, which is about the delay experienced by packets travelling between hosts located at two opposite coasts of US, the performance heavily favors implicit scheduling.

The next table illustrates that with local priority scheduling, as the number of jobs increase, the performance improves. This can be seen by observing that the idle time for each job decreases when there are more jobs. We present the actual times rather than percentages to illustrate the point.

Comparing the implicit scheduling case for three and five jobs, one finds that a process spin-waits for lesser amount of when the number of jobs is higher. This is however, not noticeable in Sor, which is computation intensive.

Message latency 300 μs, Time (in secs)
3 jobs - (Barnes+Water+Sor)
5 jobs - (Barnes+Water+Sor+Barnes+Water)

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit - 3 jobs)	4.33	12.26	40.91	4.68
Barnes (Cosched - 3 jobs)	2.97	12.26	33.57	0.61
Barnes (Implicit - 5 jobs)	3.88	12.26	76.61	4.27
Barnes (Cosched - 5 jobs)	2.97	12.26	65.13	0.61
Water (Implicit - 3 jobs)	1.92	13.59	37.12	3.60
Water (Cosched - 3 jobs)	1.79	13.59	33.56	0.46
Water (Implicit - 5 jobs)	1.41	13.59	69.56	2.42
Water (Cosched - 5 jobs)	1.79	13.59	64.81	0.46
Sor (Implicit - 3 jobs)	3.22	24.70	39.49	3.86
Sor (Cosched - 3 jobs)	3.32	24.70	31.26	2.77
Sor (Implicit - 5 jobs)	3.26	24.70	75.90	3.84
Sor (Cosched - 5 jobs)	3.32	24.70	62.53	2.77

The expectation for this result can be explained by observing the sharper difference in the next table with a larger message latency.

Message latency 3000 μs, Time (in secs)
3 jobs - (Barnes+Water+Sor)
5 jobs - (Barnes+Water+Sor+Barnes+Water)

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit - 3 jobs)	8.18	12.26	54.99	8.52
Barnes (Cosched - 3 jobs)	18.14	12.26	63.68	1.17
Barnes (Implicit - 5 jobs)	6.29	12.26	92.94	6.69
Barnes (Cosched - 5 jobs)	18.14	12.26	124.19	1.21
Water (Implicit - 3 jobs)	4.82	13.59	51.86	7.52
Water (Cosched - 3 jobs)	14.49	13.59	62.37	0.84
Water (Implicit - 5 jobs)	2.91	13.59	84.53	4.68
Water (Cosched - 5 jobs)	14.49	13.59	120.52	0.84
Sor (Implicit - 3 jobs)	3.83	24.70	54.80	5.42
Sor (Cosched - 3 jobs)	4.73	24.70	59.56	5.30
Sor (Implicit - 5 jobs)	3.97	24.70	92.39	5.63
Sor (Cosched - 5 jobs)	4.73	24.70	119.11	5.30

With the message latency kept as $3000\mu s$, as more jobs are run together, the spin-waiting time of the communication intensive processes have decreased. The same has not happened for the computation intensive Sor.

Since when more jobs are running, there are better options for the priority scheduler to select the job that uses the least amount of CPU in its time slice, the jobs that waited longer are re-scheduled.

Since the job waited longer, it is expected that its message would have returned by the time is gets scheduled. This is not so noticeable when the message latency was lower ($300\mu s$), since messages came back quicker than the time the job gets re-scheduled. The priority of processes of Sor are usually low, and hence the difference is not noticeable for Sor between three and five jobs.

Finally, to compare the advantage of using a priority local scheduler rather than a pure round-robin scheduler for implicit scheduling, we also implemented a round-robin local scheduler. The results are presented in the following table.

Message latency 400 μs, Barnes+Water+Sor (3 jobs), Time (in secs)

Job	Idle	Busy	Blocked	Switching
Barnes (Implicit-pri)	4.46	12.26	40.54	4.81
Barnes (Implicit-rr)	1.61	12.26	50.30	2.19
Barnes (Cosched)	3.47	12.26	34.68	0.63
Water (Implicit-pri)	2.54	13.59	37.99	3.37
Water (Implicit-rr)	1.10	13.59	51.23	1.61
Water (Cosched)	2.36	13.59	35.15	0.47
Sor (Implicit-pri)	3.25	24.70	40.53	3.95
Sor (Implicit-rr)	5.26	24.70	22.11	7.26
Sor (Cosched)	3.38	24.70	32.77	2.87

We notice that the blocked time for Sor is more in priority scheduling than in round-robin. This is because, Sor being computation intensive is always given a lower priority than the more communication intensive jobs, Barnes and Water. Also, for the same reason, Barnes and Water are blocked for a larger period in round-robin than in a priority based local scheduler. This, similarly, explains the reverse trend in the idle times. A communication intensive process is scheduled in tn its usual turn in a round-robin, which is later than in the case of priority scheduler, and hence blocks the process long enough to allow the message to return before it is scheduled again. Hence, its idle time decreases.

Analyzing the results, we feel that implicit scheduling is a better choice as the message latencies between processors increase. It is definitely competitive with co-scheduling in most of the above choice of message latencies and clearly better as the latencies get larger. Also, parameter for the spin-waiting phase should be so chosen so that the spin-waiting phase is not kept just less the message round-trip time (as illustrated by comparing the performance of Barnes for latencies of $50\mu s$ and $300\mu s$).

7 Conclusions

The simulation model, though approximate, has given a fair idea of trade-offs between co-scheduling and implicit scheduling. To get more exact figures, one needs to do a more detailed modeling of the underlying architecture. For example, we can take into account the dynamically changing nature of the working set size. This would be a better model, as it reflects the nature of the processes more accurately than using a fixed statically determined estimate of the process working set.

Looking at these results, we feel that the implicit scheduling technique is definitely a good choice for parallel processes, particularly when one adds the advantage of using existing schedulers that are already available in the processors.

Better performance can be obtained if we can more accurately predict the turnaround time for a message. For example, a memory request (in our model) is serviced immediately, whereas barrier messages are responded to only when all processes have sent in their barrier arrivals. However, this means that the scheduler needs to be more aware of the job characteristics than it normally is.

8 References

[1] J.K. Ousterhout. Scheduling techniques for Concurrent Systems. 3rd International Conference on Distributed Computing Systems, May 1982.

[2] A.C. Dusseau, R.H. Arpaci, D.E. Culler. Effective distributed Scheduling of Parallel Workloads. Proceedings of ACM Sigmetrics '96.

[3] P. Keleher. Lazy Release Consistency for Distributed Shared Memory. PhD Dissertation Thesis. Rice University, 1994.

[4] S.J. Leffer, M.K. McKusick, M.J. Karels, J.S. Quarterman. The Design and the Implementation of the 4.3BSD Unix Operating System. Addison Wesley Publishing Co. 1989.