Communication-Intensive Parallel Applications and Non-Dedicated Clusters of Workstations

Kritchalach Thitikamol and Peter Keleher

Department of Computer Science, University of Maryland,
College Park, MD 20742
{kritchal, keleher}@cs.umd.edu

Abstract. Time-sharing operating systems may delay application processing of incoming messages because other processes are scheduled when the messages arrive. In this paper, we present a simple adjustment to application polling behavior that reduces this effect when the parallel process competes with CPU-intensive sequential jobs. Our results showed that moderate improvement might be achieved. However, the effectiveness of the approach is highly application-dependent.

1. Introduction

Client-server applications often use busy loops to multiplex and retrieve incoming messages. Such loops might use UNIX `select`() calls to check for message arrival. If no message is pending, `select`() returns immediately and the applications may execute other work. The use of busy loops is often called message polling, in contrast to interrupt message delivery. Message polling is very simple to implement. However, in non-dedicated processors, time-sharing operating systems can inadvertently delay notification of incoming messages. This delay occurs because polling processes are inactive at the time their messages arrive. As a result, the message processing is delayed on both server and client sides and application performance is degraded.

In Section 2, we discuss such a performance problem in CVM [1], a software distributed-shared-memory system (DSM), executing on machines that have other active processes. Software DSMs are software systems that provide shared memory semantics across machines that support only message passing. CVM implements relaxed consistency models, supports multiple threads per processor, and allows thread migrations during application executions. In Section 3, we pinpoint the cause of the problem using simple client and server experiments and present our solution to the CVM problem. In Section 4, we then fully investigated implications from our results and the technique we used to reduce message delay. Finally, we presented our conclusion for similar polling structures and execution environments in Section 5.

Table 1. Characteristics of Sequential Processes.

Sequential Apps	Descriptions	Percent busy on an empty processor
grep	GNU grep program	15-20%
gcc	GNU gcc program	45-55%
infloop	Infinite loop program	100%

Previous studies of the message notification delay [2, 3] reduce its effect by applying special algorithms to re-schedule and synchronize incoming messages with polling processes. Many target message-passing parallel programs, in which synchronization delay is highly important. Significant improvements can be achieved with these approaches, but they often required modifications to the underlying operating system kernels. On the other hand, our technique modifies only the CVM application. In the worst case, our results also show significant performance improvement with our approach. The method basically allows CVM polling processes to avoid long context switches during the critical path of handling incoming messages. While the experiments we present here are specific to Linux 2.0.32 running on Pentium II's, we have confirmed that the problem exists on other operating systems, and similar approaches produce improvements.

2. CVM and Load Balancing

CVM maintains memory consistency by pulling shared data modifications from where they were created in a client-server style. CVM uses I/O signal handlers to notice incoming requests. However, when waiting for replies, CVM uses message polling to multiplex possible multiple replies and to schedule local threads.

We ran two applications, successive over-relaxation (SOR) and water molecular simulation application (Water) from Splash-2 [4], on nodes that had sequential jobs on them. The intent was to test CVM's thread reconfiguration mechanism for load balancing. Imbalance computation time observed in our experiment is a direct result from sequential jobs that compete for CPU cycles with CVM processes. By moving some of the computation to even their executions in thread granularity, CVM processes can avoid increasing global synchronization delay thus improves their overall performance in non-dedicated environment.

The machines we used are a cluster of Pentium-II 266MHz machines running Linux operating system and connected by Ethernet and Myrinet networks. The experiments basically included several runs of each application with different numbers of threads on four processors. We also ran one sequential process on the last processor. Table 1 shows characteristics of our sequential processes and their CPU-busy percentage on our Linux platform with no other processes. For each configuration execution, we measured its elapsed time after a specific sequential process started.

Table 2: Performance of CVM applications using 0-μsec timeout with sequential process on the last processor.

Appls	Active seq. processes	Elapsed time in second	
		“8,8,8,8”	“10,10,10,2”
SOR	none	6.70	8.30
	grep	8.14	8.31
	gcc	10.66	10.17
	inf-loop	13.20	11.22
Water	none	25.88	30.78
	grep	28.58	30.95
	gcc	47.98	38.82
	inf-loop	51.70	44.97

Our expectation was that reconfiguring the mapping of threads to nodes would limit the degradation caused by the sequential job running on the last node.

Table 2 shows the performance results with 32 threads in total. The third and fourth columns show elapsed time from two configurations; “8,8,8,8” means 8 threads are mapped to each processor, and “10,10,10,2” means the first three processors have 10 threads and the last has only two threads. Both used our original busy-wait polling and are labeled 0-μsec in the table. The smaller number of threads on the last processor reflect competition for CPU cycles because of the sequential process. The results are disappointing, to say the least.

We also observed that even though we were balancing application computation, the results had high variability from run to run with different type of sequential process. We would expect the “10,10,10,2” to produce similar performance regardless of the guest processes because we expected that during the execution, barriers would force two threads on the last processor to wait until all 10-thread processors finished their computations. Also, the priority of sequential processes and CVM processes were no difference. In fact, the elapsed time of the last configuration SOR in Table 2 varied from 8.31, 10.17 to 11.22 seconds. These non-uniform results imply that there must be some other artifacts that are affecting barrier imbalance besides the application’s computation.

3. Testing Message Polling with Sequential Processes

SOR’s simplicity made it easy to rule out computation variability as the culprit. Hence, the load balance must be the result of changing computation costs. In order to verify that the problem was not an artifact of CVM’s structure, we wrote simple client and server programs using UDP-sockets. Presumably, if we execute our server with the sequential processes, it would produce a relative degradation of communication performance similar to the first performance results in Table 2. More specifically, our client process continuously sends requests, blocks waiting replies, and then sends more requests. The server uses a non-blocking call to `select`() in order to implement a busy-wait loop. Calls to `select()` are made non-blocking by specifying a zero-microsecond (busy) delay as a parameter. During the experiments, we collected average message round trip time of 20,000 messages ping-ponging between client and server.

In order to explore other options that might affect our communication performance, we also tested our applications with constant timeouts passed to the `select()` calls. We tested 1, 2, 5, 10 and 20 μsec delays, and message sizes of one word and 8192 words.

Fig. 1 shows our client-server results, which we measured on Ethernet and Myrinet networks. The `select()` blocking periods are listed on the x-axis. The y-axis shows the average round trip time for each run. The results using one word messages are on the left and results using 8192-word messages are on the right. Lines in both charts represent results from different networks with different sequential processes in use shown below both charts. The "E" and "M" refer to Ethernet and Myrinet networks respectively.

The results with 0-μsec with any sequential processes confirmed our hypothesis. The average round trip time with simple busy loops degraded by a large amount in the presence of a sequential process. However, there was very little slowdown with nonzero μsec polling, regardless of the sequential processes. The busy-waiting produced better performance than non-zero μsec polling only when executing with no load. The average round trip time tended to get larger with sequential processes that generated higher percentage of CPU-busy time. The results from Ethernet and My-

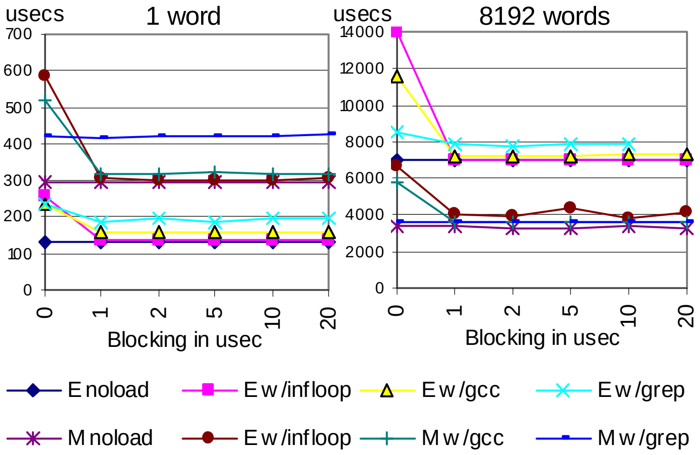

Fig. 1. The results from client-server experiment show changes in average message round trip time (*y-axis*) with different `select()` blocking periods (*x-axis*). The results using one word messages (*left*) and results using 8192-word messages (*right*) produce similar conclusion. Lines in both charts represent results from Ethernet (*E*) and Myrinet (*M*) networks with different sequential processes in use (*noload*, *infloop*, *gcc* and *grep*) shown at the bottom of the figure.

Table 3: Performance of CVM applications using 1-µsec timeout with a sequential process on the last processor and "10,10,10,2" mapping.

Appls	Active seq. processes	Elapsed time in second
SOR	none	8.31
	grep	8.44
	gcc	8.45
	inf-loop	8.44
Water	none	30.74
	grep	35.72
	gcc	35.97
	inf-loop	35.78

rinet networks showed similar patterns, but the Ethernet was faster for small messages and slower for large ones.

Based on these results, we re-ran the CVM experiments. Only the CVM process running on the node with the sequential job used 1-µsec timeouts instead of non-blocking `select()`. Table 3 shows their times in the last column. The results show slight degradation with little or no load, but significant improvements with high load, in comparison to "10,10,10,2" 0-µsec runs in Table 2.

Although we can conclude that message delay is likely the source of additional slowdown, we had yet to identify the exact mechanism of the performance problem. However, we hypothesized that the difference results from OS scheduler behavior.

4. Time Sharing Behaviors

Intuitively, it is easy to understand that message replies may be delayed because polling processes are suspended when the message arrives. The message cannot be seen by the application until the underlying operating system re-schedules the polling process. In order to verify that this is the case in our experiments, we re-ran the client-server experiments and recorded individual round trip times to see whether there were any abnormal message-delays with the infinite loop process. We selected the infinite loop program because it always uses up its time slice or quantum time and the scheduler behavior becomes fairly predictable. The client and server experiments were again repeated only with 0-µsec and 1-µsec blocking time in the `select()` call.

In addition, we modified the Linux kernel to report the current time counter of the server process. The time counter indicates how much time remains in the currently running process's time slice. Each time tick in time counter is about 10 milliseconds worth of execution time. The Linux priority-based scheduler preemptively makes a context switch when the current process's time counter becomes zero, or the process explicitly yields to another process often through UNIX system calls.

Fig. 2 shows individual round trip time of 150 messages listed on x-axis in the top chart and corresponding values of remaining Linux time counter of the server process at the bottom chart. The results confirmed that the 0-µsec server was switched out for full quantum time, approximately 0.2 second by Linux default, (because of the infinite loop program) more often than 1-µsec server. For example, the time counter of 0-µsec server went to zero more often than 1-µsec in Fig. 2. Note that the round trip time of the three spikes in the top chart was between 0.1 and 0.3 seconds, which is much higher than the normal round trip time of 0.029 second.

Although the numbers help us understand the source of the problem, we still need to understand exactly how the 1-µsec blocks change the scheduling behavior. It turns out that the scheduler periodically switches out the 0-µsec server because it exhausts its ticks. However, the 1-µsec block causes the kernel to temporarily yield the CPU to another process during the delay time. By watching the Linux scheduler during our 1-µsec server experiment, we observed 1505 context switches bouncing between server process and infinite loop process, comparing to only 21 context switches with 0-µsec. However, the 1-µsec switches are only for approximately the duration of the 1-µsec timeout. Additionally, the other process is usually the sequential job, which quickly uses up its ticks. The scheduler then adds new ticks to both the sequential job and all other jobs. This latter category includes the CVM process. Hence, the CVM process rarely exhausts its ticks, and rarely is swapped out for an entire time slice, resulting in better performance. Basically, the one microsecond timeout in our server experiments improves the chance of no full context switches forced by OS scheduler. The poor performance of 0-µsec server with a sequential process happened because it was inactive for full quantum time. The delay eventually adds up and creates abnormally large round trip times, especially in highly communicating processes.

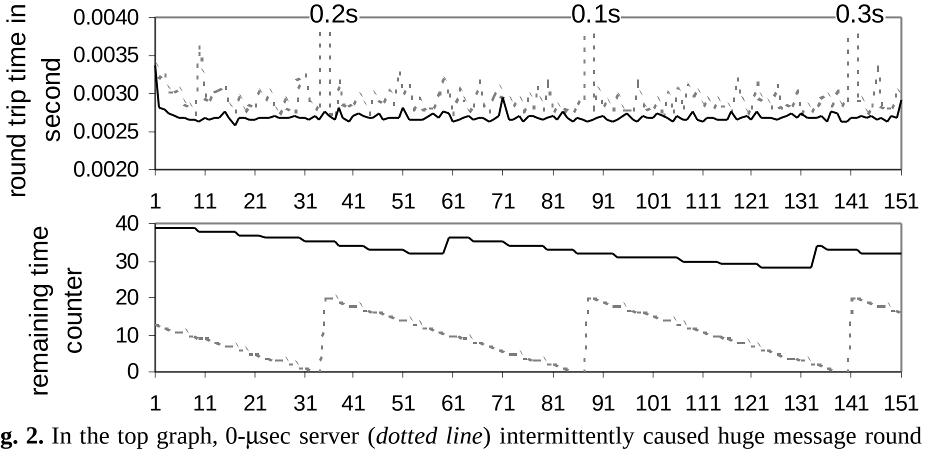

Fig. 2. In the top graph, 0-µsec server (*dotted line*) intermittently caused huge message round trip time while 1-µsec server (*normal line*) did not. The bottom graph showed that the 0-µsec server (*dotted line*) was swapped out more often than the 1-µsec server (*normal line*) as the OS counter of the 0-µsec server became zero frequently.

Although this behavior is clearly tied to specific implementations, we obtained similar results when ran the same experiments on an IBM-SP2 parallel machine running AIX 4.2. The AIX results also showed less variability in the “10,10,10,2” tests with 1-µsec blocks and sequential processes. The 1-µsec elapsed times were better with the infinite loop sequential program, but not GCC. Note that the sequential processes produced different CPU-busy percentages between AIX and Linux operating systems, except only the infinite loop program that maintains 100% busy.

We realized that the 1-µsec block was less effective with less resource-hungry sequential processes on both platforms, e.g. with grep program. In Linux, this is because time counters of lightly busy sequential processes become zero less often. The OS scheduler adds time counters only after a process exhausts its time, and it is likely that less intense sequential jobs use all of their quantum less frequently. Therefore, the blocking technique produces less improvement with less busy jobs. Fortunately, these jobs also produce less slow down to begin with.

Nonetheless, to avoid degradation with lightly busy processes, we suggest our approach be used in dynamic fashion. For example, CVM processes should be able to decide 0-µsec or 1-µsec blocking parameter based on CPU-busy percentage of adversary processes. Although we have not tested our approach with all available operating systems, our experience suggests that for 50-100% busy guest processes, 1-µsec blocking be worth trying.

Our further research reveals that modern operating systems, including Linux and AIX, similarly implement process scheduler with non-fixed priority. The scheduler basically recalculates running processes’ priority value at each clock interrupt, which may not be equal to a period of time slice. Therefore, a process may lose its CPU control because its priority value is inferior to that of another dispatchable process. Consequently, we believe that the similar results from our technique can be obtained with other operating systems that use non-fixed priority scheduling. Ultimately, the idea is to avoid OS scheduler to penalize communication-intensive processes with a lot of full time-slice message delay, which have been proved to be expensive for parallel processing in non-dedicated environments.

5. Conclusions

We have shown that we can reduce the impact of competing sequential jobs by changing application code, without touching the operating system scheduler. Our technique consists of using blocking message polls when competing with CPU-intensive jobs. The result is that we reduce the chance of the parallel job being switched out for a long period of time. Its effectiveness, however, depends largely on the level of CPU usage of the local sequential processes. For best results, the technique should be adjusted dynamically. We confirmed the applicability of our method with simple client-server and CVM load balancing experiments on both Linux and AIX operating systems, and the results showed encouraging performance improvement.

References

1. P. Keleher. *The Relative Importance of Concurrent Writers and Weak Consistency Models*. in *Proceedings of the 16th International Conference on Distributed Computing Systems*. 199691-99. Hong Kong: IEEE.
2. A.C. Dusseau, R.H. Arpaci, and D.E. Culler. *Effective Distributed Scheduling of Parallel Workloads*. in *Sigmetrics'96 Conference on the Measurement and Modeling of Computer Systems*. 1996.
3. P.G. Sobalvarro, *et al. Dynamic coscheduling on workstation clusters*. in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. March 1998.
4. S.C. Woo, *et al. The SPLASH-2 Programs: Characterization and Methodological Considerations*. in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. June 199524--37. Santa Margherita Ligure, Italy.