Thread-Parallel Job Performance in a Time Sharing Batch Environment on Origin 2000 Systems

David McWilliams
National Center for Supercomputing Applications
605 E. Springfield Ave.
Champaign, Illinois 61820 USA

ABSTRACT: *At the National Center for Supercomputing Applications (NCSA), we found that thread-parallel, gang-scheduled Origin 2000 jobs consumed more than 5 times as much CPU time when the load average on a system is high (96 on a 64-processor system). We worked with Silicon Graphics (SGI) as they made changes in the IRIX process scheduler to fix the problem. We will discuss improvements to the scheduler and the general problem of scheduling parallel jobs when processors are over-allocated. NCSA's Load Limiter ensures that the load average does not exceed the number of processors. Parallel jobs now have much more consistent performance.*

Introduction

NCSA SGI Origin Production Environment

NCSA's production environment includes 9 SGI Origin systems with a total of 512 processors and 160 Gbytes of memory. A 32 processor Origin is available for interactive use. The remaining 8 systems are reserved for batch use. NCSA uses lsbatch, part of the Load Sharing Facility (LSF) from Platform Computing Corporation as its batch system. There are about 26 batch queues, categorized by resource requirements (memory and CPU), user type (academic or Industrial), and type of queue (time-shared, debug, or dedicated). Since it is a national center, NCSA has thousands of users that are geographically dispersed throughout the U.S. and the world. Our users represent a large variety of disciplines, types of codes, and programming models.

The problem

In the summer of 1997, users reported widely varying CPU times for the same job on NCSA's Origin systems. NCSA requires users to put CPU limits on their jobs. When the CPU limit is exceeded, the job is killed. Some users reported that the same job ran fine sometimes and other times it was killed because it exceeded CPU limits. Other users reported that a job would use 10 CPU hours on one run and 40 hours on the next. The wall clock time of each job was proportional to the CPU time, so it was difficult to predict when a job would finish. This was making a big impact on user productivity and satisfaction.

The Testing Process

The Maxwell code

NCSA obtained a code that demonstrated the problem particularly well. According to the user, the *maxwell* code "performs a time-domain simulation of Maxwell's equations in a dielectric on a rectangular grid." It was compiled with version 7.1 of the Fortran 90 compiler and parallelized with the –pfa option. Thus, the code uses an MP thread-parallel programming model. The observed performance was good in a dedicated environment when the load average was low. The user reported that the code typically ran in 20 CPU hours, but occasionally took over 50 hours.

Testing Process

We obtained the maxwell code from the user and ran it in a series of tests on a dedicated 64-processor Origin system running IRIX 6.4. We ran one to six copies of identical 16-processor jobs, varying the system load average from 16 to 96. All the jobs had *gang scheduling* enabled. With gang scheduling, all threads are scheduled on processors at the same time so the threads are synchronized with each other. We computed the average CPU and wall clock time for all the jobs in each run.

Results

Initial Results

Figure 1 shows the results of the test. We show the average CPU seconds per job, observed and ideal. In the ideal case, we naïvely assume the average CPU time for each job

will stay the same, regardless of the system load average. Figure 1 shows that the average CPU time grows exponentially when the number of active threads exceeds the number processors.

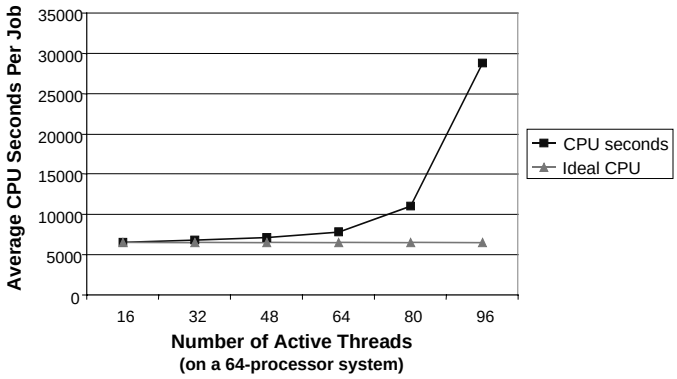

Figure 1. Average CPU seconds per job for one to six copies of thread-parallel gang-scheduled jobs.

Figure 2 shows the wall clock (REAL) time for the same jobs that were shown in Figure 1. In estimating the ideal time, we expect the wall clock time to increase when the number of active threads exceeds the number of processors since the threads have to share processors. As a simplifying assumption, we ignore cache effects that occur when a thread is moved from processor to processor. As with the CPU time, the average wall clock time grows exponentially when the number of active threads exceeds the number processors.

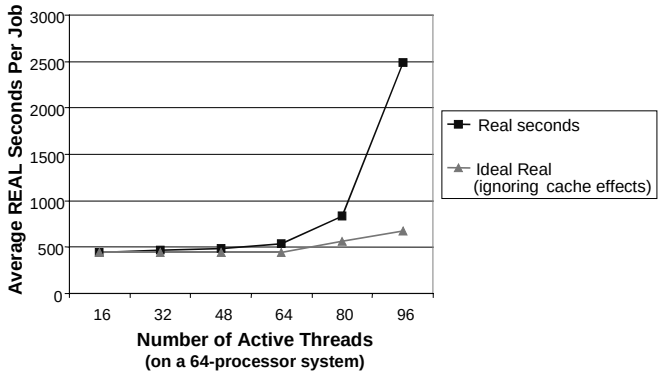

Figure 2. Average wall clock (REAL) seconds per job for one to six copies of thread-parallel gang-scheduled jobs.

Observations

Using *perfex*, we observed the inefficient use of cache when there are more threads than processors. The movement of threads from processor to processor can explain this effect. The observed degradation in performance due to cache misses is on the order of 15% to 20% on 96 active threads. It cannot explain the large exponential effect in figures 1 and 2.

Using *ssrun*, we observed that the threads were spending a lot of time in barriers when there were more threads than processors. This was unexpected behavior and it led us to believe there might be a problem with gang scheduling in the IRIX 6.4 scheduler.

SGI Response

We shared our results and observations with our on-site SGI applications engineer, K.V. Rao, who gave our results and the maxwell code to SGI engineers in the Resource Management Group. They made changes to the IRIX 6.4 scheduler, while NCSA served as a beta site. These changes were included in patch 2536, which became available in January 1998. It is now included in patch 2890.

The patch ensures that two threads in the same gang will never be scheduled onto the same processor.

Here is the analysis of the problem from the SGI engineer. The maxwell code had poor memory placement. Rather than distributing the memory across various nodes, all the memory was in one node. The scheduler tries to schedule threads on processors close to where the memory is. So the scheduler was putting multiple threads on the same processor. You can imagine that at times, one of the threads was waiting on a barrier for the other thread, while both were competing for the same processor.

While the programmer could have addressed the memory distribution problem, it is clear that the scheduler should not be placing two threads in the same gang on the same processor.

This is a great example of a customer and vendor working together to solve a problem. Our users are satisfied and SGI has a better product as a result.

Results after Patch 2536 was installed

Figure 3 shows the results of the test after patch 2536 was installed. The scheduler is doing a much better job scheduling the threads of the gang-scheduled process.

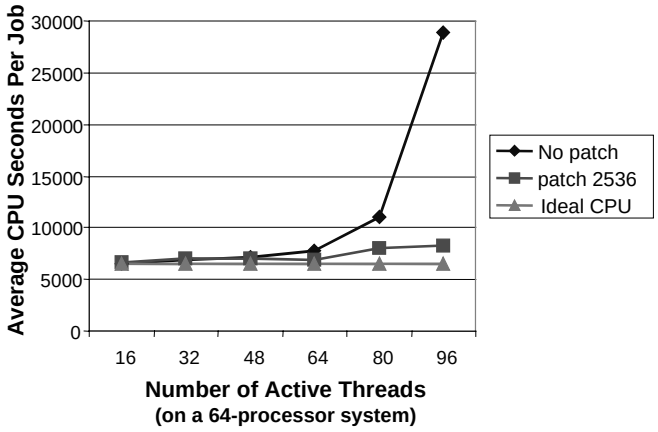

Figure 3. Average CPU seconds per job for one to six copies of thread-parallel gang-scheduled jobs after patch 2536 was installed.

Figure 4 shows the wall clock time for the same job mix. It shows a comparable improvement in wall clock time.

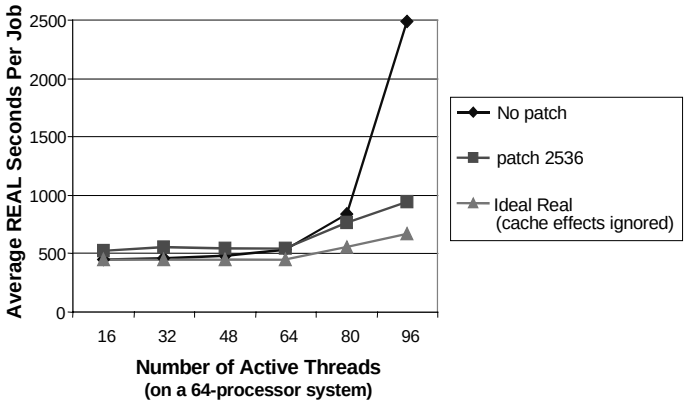

Figure 4. Average wall clock (REAL) seconds per job for one to six copies of thread-parallel gang-scheduled jobs after patch 2536 was installed.

Environment Variables

SGI suggested we try experimenting with various environment variables to modify the way the scheduler works. We tried various values of the following variables:

- $MPC_GANG
- $MP_SUGNUMTHD
- $MP_SCHEDTYPE
- $MP_BLOCKTIME
- $_DSM_MUSTRUN
- nodemask

Only two had any significant effect. $MPC_GANG turns gang scheduling on and off. With gang scheduling, all threads are scheduled on processors at the same time so the threads are synchronized with each other. Gang scheduling is on by default under IRIX 6.4, and it is critical for good performance with MP thread-parallel codes when dynamic threads are not used.

Dynamic Threads

The variable $MPC_SUGNUMTHD enables *dynamic threads*. As the program runs, an asynchronous process monitors the system load. Each time the program enters a parallel region, the number of threads is adjusted based on the load average. If the system load is high, it decreases the number of threads. If it is low, it increases the number of threads.

When dynamic threads are enabled, the number of *active* threads is always less than or equal to the number of processors. For example, when 6 jobs are running, they are only using about 11 threads each on a 64-processor system.

Dynamic threads make a big difference in performance, because it makes the scheduler's job easy. The scheduler can put a thread on a processor and leave it there. Figures 5 and 6 show that when all the jobs use dynamic threads, the system load average is controlled, and thus the scheduler can efficiently place threads onto processors without contention. The performance of the jobs with dynamic threads is very close to ideal performance, regardless of whether gang scheduling is used.

With the patch, there is a small increase in wall clock time when gang scheduling is used. If dynamic threads are used, gang scheduling should be turned off to achieve the best performance.

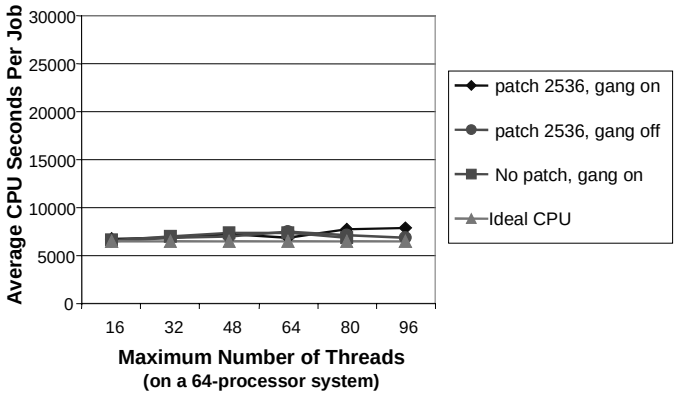

Figure 5. Average CPU seconds per job for one to six copies of thread-parallel jobs with dynamic threads enabled.

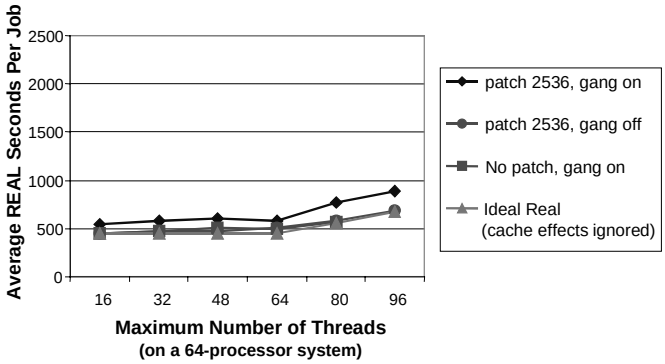

Figure 6. Average wall clock (REAL) seconds per job for one to six copies of thread-parallel jobs with dynamic threads enabled.

If all jobs were MP parallel and all used dynamic threads, you would achieve the best system throughput. Unfortunately, the use of dynamic threads cannot be enforced. In a mixed environment, those users that use dynamic threads will pay the price in slower turnaround time.

Dynamic threads are not useful for benchmarking, because they produce unpredictable wall clock times (due to the variable number of threads). Dynamic threads are not available for programming models other than thread-parallel.

Dynamic threads will be on by default under IRIX 6.5, with gang scheduling off. This provides the best system throughput. If dynamic threads are turned off, gang scheduling

is automatically turned on. This provides the next best system throughput on a heavily loaded system.

General Problem

The patch solves the problem of CPU and wall clock performance on MP thread-parallel jobs when gang scheduling is used. Unfortunately, NCSA's actual job mix is more complex. We have a mixture of programming models: MP, HPF, SHMEM, POSIX threads, MPI, and PVM.

When running MPI jobs, for example, we have seen problems similar to what we saw with the maxwell code before the patch was installed. The reason is that the scheduler does not know that the MPI processes are part of the same job. So it cannot help scheduling multiple MPI processes onto the same processor.

The only way to generally solve the problem is to control the system load average to make the scheduler's work easier. Dynamic threads do that, but only for MP thread-parallel jobs.

NCSA's Solution

NCSA Load Limiter and Job Process Monitor

NCSA extended the LSF batch system by creating the Load Limiter and Job Process Monitor, which were written by Michael Shapiro (mshapiro@ncsa.uiuc.edu).

The basic idea of the Load Limiter is to require users to specify the resources (number of processors, memory, and CPU time) that their job will use. The Load Limiter ensures that jobs are scheduled only when the resources required are available. The Job Process Monitor ensures that jobs do not exceeded their stated requirements as well as reporting the actually resource usage for the job. If the job exceeds its declared requirements, it is killed.

NCSA's Load Limiter uses LSF's resource manager. Each host system has a pool of processors. When a job is started, the number of available processors is reduced by that amount. A new job will not be started on a host unless there are enough processors available in its pool. Memory works in a similar way. One queue may span several host systems. When a new job is submitted, LSF selects a host that has enough processors and memory.

With the combined effect of the IRIX patch and the Load Limiter, our users have been seeing much more consistent CPU and wall clock times.

There is a description of NCSA's batch system at http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000/Doc/Jobs.html.

Issues with NCSA's Solution

There was a problem where jobs that require a lot of processors wait forever because there is a constant stream of jobs that require few processors. To solve this problem, some hosts systems are designated "large job" systems. If there are any large jobs in the queue, those systems will not start any smaller jobs.

One problem with the Load Limiter is that it requires users to specify new limits on every job. If the limit is too low, the job is killed. But if the limit is too high, system resources go idle. We want the user limits to be as close to the high water mark as possible.

There is also the issue that the actual use of resources (both memory and number of processors) may vary over the lifetime of a job. For example, a benchmarking job may use 1, 2, 4, 8, 16, 32, and 64 processors. The user must set the limit to 64, even though most of the time the job does not need that many processors.

The Job Process Monitor reports the high water marks for memory and the number of processors after a job has finished, so users running production jobs should be able to give very accurate limits.

To increase system utilization, we want the scheduler to be able to handle some over-allocation of processors and memory without causing system throughput problems. This is certainly possible now when all the jobs are MP parallel and either dynamic threads or gang scheduling is used, but NCSA's job mix is more complex.

Future Plans

Message Passing Jobs

We would like to evaluate MPI jobs and even a heterogeneous job mix, such as MPI and MP parallel jobs. At this time, the scheduler has no knowledge that multiple processes of an MPI or PVM job are part of the same job. Thus, we expect that the performance of MPI and PVM jobs will be poor whenever the system load average is high. We think it would be useful to have an equivalent to gang scheduling and dynamic threads for other programming models, such as MPI.

IRIX 6.5 Scheduler

We would like to evaluate the job mix under the IRIX 6.5 scheduler. Engineers in SGI's Resource Management Group have told us that it is much improved over IRIX 6.4.

Miser

Miser is a resource management facility with a batch process-scheduling component that will be included with IRIX 6.5. Miser allows the system administrator to allocate physical processors and memory to a job, achieving close to dedicated performance. Miser provides kernel support for limiting the load average of a system. We are exploring how Miser will help us control the load average of NCSA's Origin systems.

Miser's usefulness to NCSA really depends on how it is integrated into LSF. It's not useful to us as a stand-alone system. NCSA is talking with Platform Computing and SGI to give feedback on how we think Miser should be integrated into LSF. We expect that Miser will be integrated into LSF later this year.

Conclusion

It is extremely important to control the system load average to run parallel jobs efficiently on Origin systems.

If all parallel jobs use the MP programming model, then it's best for all jobs to use dynamic threads. The next best alternative is for all jobs to use gang scheduling.

NCSA will continue to use the Load Limiter for the immediate future, and we are planning to incorporate Miser into our batch system after it has been integrated into LSF.

Acknowledgements

K.V. Rao, R. Ananthanarayanan, and Nawaf Bitar of Silicon Graphics have been extremely helpful in this project. The author wishes to thank Michael Shapiro, Faisal Saied, and Rick Kufrin for their assistance with this paper.

About the Author

Dave McWilliams (davem@ncsa.uiuc.edu) is a member of the Performance Engineering and Computational Methods group at the National Center for Supercomputing Applications (NCSA) at the University of Illinois.