Multi-CPU Pools in an NQS Environment

Tim Folkes, CCLRC Rutherford Appleton Laboratories
Roger Evans, CCLRC Rutherford Appleton Laboratories
Mike Armstrong, Cray UK

ABSTRACT: *Utilising the resources of a J-932 with a wide range of batch work is very difficult using only the standard NQS controls. Jobs that need most of the system resources in memory or temporary disk space should be able to access most of the CPU's when needed without holding on to idle processors at other times. Dedicated run times and overnight queues are not very popular with users who wish to develop applications throughout the day. Our new multiple CPU job classes use the UNICOS real time scheduling features to create pools of typically 8 or 16 CPU's for parallel jobs. The parallel jobs have a high probability of finding all the processors they need while returning under used processors to the normal scheduling mechanisms.*

Introduction

Background.

History

From the time of our use of Cray X-MP/416 and Y-MP/8128 machines it has long been a problem to schedule jobs which need a large fraction, but not all, of the machine resources of CPU's and memory. To schedule these jobs in a dedicated mode wastes a fraction of machine resources: to schedule them along with a variable work load results in unpredictable performance and little or no gain to the user who wishes to see some reward for the effort they have put in to parallelise their codes.

Problem

On a 32 CPU J-90 the problem is greatly compounded, all the more so by the difficulty of getting good performance out of a large fraction of the CPU's. With a single CPU performance not much better than a workstation, it is only through effective parallel processing that a user can see a significant benefit in elapsed time and the system managers can allow the efficient running of large memory jobs.

Aims

Our ultimate aim is to be able to schedule a mix of batch work that will have the choice of a single CPU, or multiple CPU's, probably 4, 8 or 16. Most of our testing so far has been with an 8 CPU queue. The target for the "multi" queues is that on average they should get in excess of 90% of the requested number of CPU's for the duration of their execution. This paper concerns itself with the implementation of this preferential scheduling through real time processes and the achieved performance on a busy machine with a mix of interactive and batch work. At a future CUG we hope to report on the scheduling issues for a mix of these multiple CPU queues to give a good performance and to control memory allocation and swapping.

System

Configuration

Hardware

The machine we now have installed is a J-932/4096. We have 216 Gbytes of Cray disk (6 DDS30's) and 108 Gbytes of third party SCSI disks attached to the system.

A STK 4400 silo is SCSI attached to the system with four 4480 drives. These are used exclusively for data migration and system backups. Any user tape activity is handled via the Virtual Tape Protocol into an IBM 3494.

The J-90 has two FDDI connections and one ATM connection. As yet the ATM interface has not been used.

Software

The J-90 is running -

- UNICOS 9.0.2.1
- OSV 2.0
- dmf 2.4.2
- cf77 6.0.4.28
- cc 4.0.3.22
- f90 1.0.3.4

- CC 1.0.3.3
- Programming Environment 2.0
- MPT 1.0

Machine Load

User Profile

The J-90 has 1232 registered users. Of these only about 500 are active at any one time with up to 100 login sessions in the afternoon. The range of disciplines that the users cover is shown below.

Due to this wide range of disciplines, the job mix changes by the hour; let alone by the day. This makes trying to predict how a job will behave on the system almost impossible.

Job Profile

We have tried to encourage users who use large amounts of the machine's resources to try to multi/auto task their work. On a loaded system this does not give the performance improvements that we had hoped. The job will only be allocated CPU's that happen to be free at the time.

NQS queues have been dedicated to certain classes of users and were particularly affective for scheduling the whole of the Y-MP where speedups of 7.6 times over one CPU were readily attained. Running the "WHOLE" queues overnight is much less satisfactory on a 32 CPU machine and even more of a problem after the loss of 24 hour operations cover.

System Performance

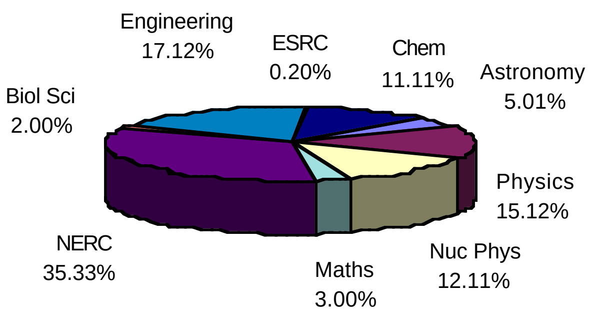
The graph below shows a typical day of CPU utilisation on the J-90. The work load tends to drop at night as the batch work is drained from the system, with an increase in load as the day progresses and new work is submitted.

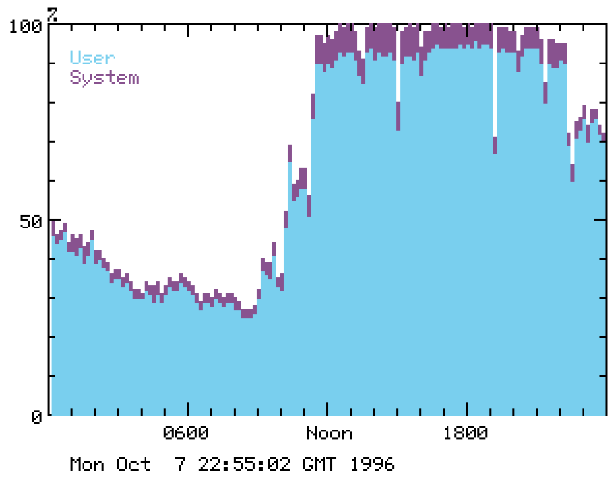
As can be seen, at peak times the system does get very busy. This graph was taken while we were still running UNICOS 8.0.

Solution.

Requirements

Against this background we approached Cray for some proposals as to how we could guarantee a user's job that is well multi-tasked, would have all the CPU resources it needed, when





it needed them. This should not have any adverse effect on the overall performance of the machine or severely impact other jobs and interactive performance.

As the machine has 32 CPU's, it would be useful to 'isolate' 8 or 16 CPU's for these high performance jobs and leave the remaining CPU's for other less parallel work. This has the added benefit of enticing users to optimise their code as they would see real benefits from using these dedicated CPU's.

To make the quest more interesting it should be remembered that J-90 systems are binary only, so there is no source code to be "hacked". We also wanted to keep the number of user exits used to a minimum to ease any future upgrades to the operating system.

However, it was clear that dedicating CPU's to particular jobs was not going to be an efficient use of resources. There is no hard criteria for deciding what qualifies as a high-performance job, so it would be possible to find users running jobs on these dedicated CPU's that did not parallelise well. Some system had to be found scheduling the CPU's so a well multi-tasked job wants a CPU it gets a CPU no matter what other work is running. This way, we can provide near-dedicated scheduling for particular jobs but without wasting CPU cycles.

Options investigated

The simplest way of increasing the potential for a job to be scheduled by UNICOS is to decrease its 'nice' value. This in theory should raise the priority of the job so that it gets preference over other jobs in the system. Some testing was done on this, but it was found that whilst it does provide some improvement, these jobs did not receive the full CPU resources that they wanted, especially when the machine is busy.

Another option investigated was to modify the scheduling mechanism in UNICOS to schedule certain jobs before others. This could have been combined with a local NQS user exit to 'tag' jobs in certain queues so that all jobs running in these queues get dedicated resources. This would ultimately be the best solution, but as we mentioned earlier, we have no source code.

The solution we eventually chose was to use the UNICOS real-time processing feature.

Design Details

UNICOS real-time processing allows jobs to set themselves up as ‘real-time’ processes and these jobs are then guaranteed by the kernel to receive as much CPU time as they require. It works by having a separate kernel run-queue for real-time processes which the kernel always looks at first when it comes to schedule work. When all of the processes in this real-time run-queue have been satisfied, the kernel will then schedule work from the normal run-queue.

This means that real-time processes will get priority over any other processes when the kernel is deciding which processes get a CPU, and which ones do not. This also has the advantage that if a real-time process were to go to sleep, for example, it does not ‘hog’ a CPU until it wakes up -- the CPU is returned to the system and can be used to run another process. However, when this real-time process does wake up and requests a CPU, it will immediately get one.

With these requirements in mind, we decided to create special NQS queues that users could submit jobs to, and only these queues would be allowed to use the real-time feature. These queues specified the number of CPU's the jobs contained therein would use, so that standard NQS queue complexes and scheduling criteria could be used to limit the amount of resources this special-case work could use. For example: we currently have a queue called MULTI8 that is open to certain users and sets any jobs submitted to that queue to use 8 CPU’s. As NQS has no concept of the number of CPU's a job in a queue is using, restricting the special-case queues to a fixed number of CPU's means that normal NQS resource management controls can be used. This is to make sure that only a certain maximum number of CPU’s are running real-time work at any one time. This is very important -- if the machine was to end up with real-time processes running on every CPU, the machine would appear to 'hang' until one of these real-time processes finished or went to sleep and released a CPU.

Real-time processing is not available to normal users though. Only users with the appropriate privileges can set up a process as a real-time process. To get around this, a ‘wrapper’ program that has the appropriate privileges is used. Users must submit their jobs in such a way that they execute this wrapper program and pass the name and arguments of their real program as arguments to the wrapper program.

Wrapper functionality

The wrapper program is based on the UNICOS command `ded`(8) (our version is called `gded`) and works as follows. After processing the command line arguments, it checks to see that this program has the appropriate privileges to be able to set itself up as a real-time process. In particular, on a non-MLS system or PRIV_SU system, this wrapper program must be run as root, and must therefore setuid to root. The program also checks to see if we are running on a guest UNICOS-under-UNICOS operating system as real-time processing does not work on a guest OS.

Next the wrapper checks to see if it is running as a batch job, and if so, which NQS queue it is currently running in. A configuration file is then examined to see if the current queue name is a valid one for this wrapper, and if so, how many CPU’s should it be using. The environment variable NCPUS is then set with this value, overriding the previously set value.

The program then sets itself up as a real-time process. Then the effective userid is changed back from root to the real user. The programme then calls exec(2) to start the user job over the wrapper program, thereby keeping all of the real-time privileges. The user program will then run as normal, but will received all of the CPU resources it requests, regardless of the amount of ‘normal’ work in the system.

gded

The way this works for the user is that we have set up a series of NQS queues. These are imaginatively named MULTI4, MULTI8 and MULTI16. The access to these queues is restricted. The user has to prove that the job can auto/multi task well enough to merit running in the queue.

Jobs in these queues should get 4, 8 or 16 CPU’s respectively. The `gded` programme is run before the executable in the style of `hpm`. If a user does forget to include the gded directive then the job will run as a normal job. This is not a problem except that the job is blocking an execution slot for a job that could usefully utilise the CPU’s.

Results

The following results were obtained by running a multitasked code in each of the three queues. The job just runs the binary first as a normal “user” process, followed by ja to print out the multitasking information. It then runs the same binary as a real time process.

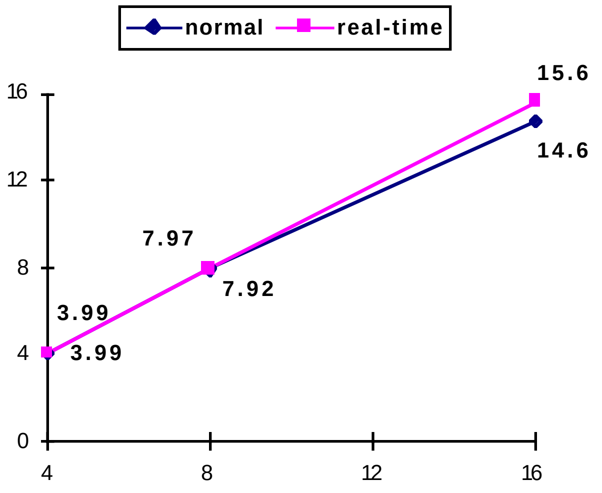
The binary is an almost ideally balanced application that solves a set of coupled wave equations in frequency and one space dimension. It consists of autotasked outer loops surrounding vector inner loops and is used locally as a standard test of machine performance.

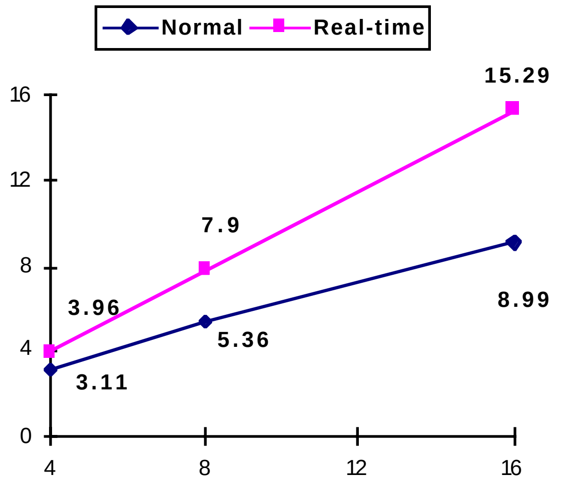
“Quiet” System

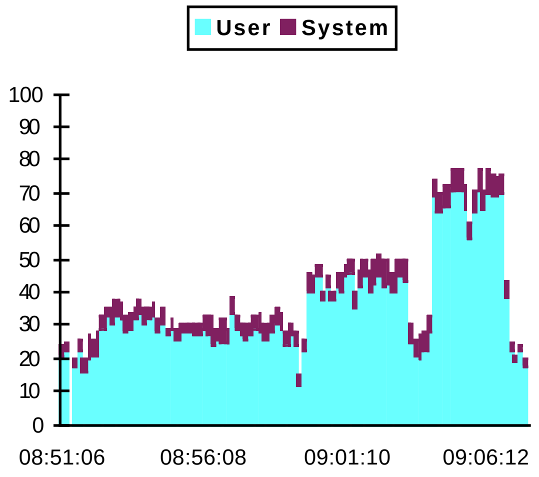
The first example was run in the morning after the user work load had drained overnight and before the users had come in and started to flood the system with the days work.

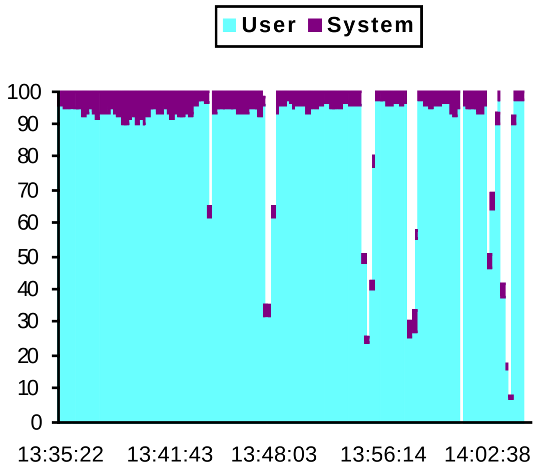
The first graph shows the number of requested CPU’s against what was delivered for the normal and real-time processes. The data points are the average number of concurrent CPU’s as reported by `ja` for the job in each of the three special queues.

The next graph is of the `sar` figures for user and system CPU time for the period that the jobs were running.









It is fairly obvious when each of the jobs was running.

Loaded System

The next set of data was taken while the system was heavily loaded. The details are the same as for the quiet system.

The dips in the above graph were due to the system going idle. This coincided with the %waitio going up and the gded process starting and finishing.

Problems

There have only been a few problems with the system so far. One problem is that while the real-time processes are running, console messages are disabled. This includes the message saying that console messages are disabled. The message saying that console messages are enabled are printed. The messages are written into the relevant syslog files if needed. There does not seem to be any way round this without source code.

The increase in waitio time seen in the sar graphs above was only noticed while the data was being collected for this paper. So far we do not have any reason for this anomaly.

We have had one user using `gded` with a programme that micro-tasks. It also calls some system library routines that are autotasked. The job was getting 15 real-time processes instead of 8. The gded programme does not limit this use of CPU's since it controls only the autotasking performance via NCPUS.

As mentioned earlier, if a user forgets to include the `gded` directive the job will still run, but as a normal process. We have had at least one user use this as a way to by-pass our usual job scheduling. If this becomes a serious problem we may have to look at using some of the NQS user exits to perform further checks on the job.

This method of scheduling does have the disadvantage that you are limited in the number of jobs of this type that you can run. We have not tried running more real-time processes that you have CPU's, but this would probably hang the system until the real-time processes had finished. Some guard against this will have to be put into NQS via queue or complex limits

Summary

We have been very pleased with the way this command has worked. Those users who have made the effort to multi-task their code are now able to make better use of the system. What we still need to do is to bias our accounting system so that we reward the MULTI users. If the job uses 8 CPU’s we may charge them for as little as 4 CPU’s as an incentive.

Acknowledgements

We would like to thank Cray UK for their assistance in this project.