Concurrent Computing – Architectures, Languages and Techniques
B. M. Cook (Ed.)
IOS Press, 1999

Supercomputing Resource Management – Experience with the SGI Cray Origin 2000

Kathryn M. MEASURES
Jeremy M.R. MARTIN
Robert C.F. McLATCHIE
Oxford Supercomputing Centre
Wolfson Building
Parks Road
Oxford OX1 3QD, UK
http://www.osc.ox.ac.uk

Abstract. The Oxford Supercomputing Centre OSC[1] was established in April 1998 to provide high-performance computing services to a consortium of Oxford University research groups. The main computer resource, an 84-processor SGI Cray Origin 2000 known as *Oscar*, is being deployed in a wide variety of research studies covering biological, medical, chemical, mathematical, physical and engineering topics (including parallel computing itself).

In this paper we shall describe the queueing and accounting mechanisms we have developed to facilitate effective use of this powerful resource. We shall also describe innovative work in progress to optimise the performance of the machine, using simulation and genetic algorithms.

1 Introduction

The Oxford Supercomputing Centre provides parallel computing facilities and consultancy to a consortium of Oxford University scientific research groups. Its mission is to promote and support multi-disciplinary research in the application of high performance computing. At the time of writing, nineteen research groups, from a wide variety of scientific and mathematical disciplines, are actively using the service.

The main resource of the OSC is *Oscar*, an 84-node SGI Cray Origin 2000[1], which has a peak computing capacity 32 GFLOPS.

The Origin is a *CC-NUMA* machine (Cache-Coherent Non-Uniform Memory Architecture). Although its memory is physically distributed between the various processor boards, which are linked by a hypercube routing network, it appears to the programmer to be as a single shared memory machine. However there are considerable variations in memory access times – from zero to over a hundred processor clock cycles[2]. This means that in order to use the Origin effectively one must pay great attention to memory management. Otherwise pathological problems such as *cache-thrash* are likely to appear.

The Origin uses the MIPS R10000 processor. This *superscalar* processor is capable of initiating four separate pipelined instructions per clock cycle. Each pipelined instruction

[1]OSC gratefully acknowledges the support of the Higher Education Funding Council for England (HEFCE) Joint Research Equipment Initiative 1997 and Silicon Graphics UK Ltd

[2]There are six levels of data location relative to a given processor, as follows: on-chip registers, primary cache, secondary cache, main memory of local node-board, main memory of remote node-board, cache belonging to another processor.

Figure 1: The SGI Cray Origin 2000

takes four cycles to execute. Potentially this means that each processor may execute upto sixteen instructions in parallel. Clearly the various instructions that a particular processor is executing at a given time need to be *independent* of each other, to ensure program correctness.

Hence there are two major challenges in developing codes for the Origin 2000 in order to keep the processors fully occupied as follows:

- Minimise memory access delays with careful memory management.
- Exploit superscalar properties of processors using fine-grained parallelism.

Much sophisticated optimisation is performed by advanced compilation technology, but the compilers cannot do it all. We find that to program the Origin effectively requires some understanding of its architecture.

At OSC there is a further complication to consider. The *Oscar* supercomputer is a heavily loaded shared resource. In practice the individual processes that constitute a user's program might have to compete with others to get scheduled. When a process finally gets a time slice it may find that its cache has been dirtied by another job. It may also find that its parallel partners have been descheduled so that it is forced to wait for communication. (This waiting is often performed using an inefficient spin lock.) It may also find that the processor interconnection network is clogged up with noise from other programs.

Under these circumstances it becomes virtually impossible to use the machine effectively. Any good work done in optimising one's code may be immaterial. What is needed is some means of getting hold of a cluster of resources and retaining exclusive access to them for the duration of a run. Fortunately this is offered by SGI's new 'Miser' scheduling system[2]. In this paper we shall describe how we have used this facility, along with the NQE spooling software[3], and job accounting, to build a fair and effective job management system for the *Oscar* supercomputer.

The rest of this paper is organised as follows. In section 2 we specify our initial requirements for job scheduling on *Oscar*. Then we describe the available software components from which to build such a system. In section 3 we explain, in detail, how the system has been constructed. Section 4 explains the innovative approach to performance optimisation that is being employed to tune *Oscar*. This is based on the technique of *genetic algorithms*. We are working towards running a series of experiments to *breed* an optimal set of system parameters.

2 Requirements for the OSC batch queueing system

Effective resource management on any system is not solely the result of implementing restrictions as to the number and size of jobs that can run on a system at any one time. Edu-

cating the end user on how to get the most out of their code, and giving an understanding of the system as a whole, in terms of the underlaying hardware, and the way in which it is managed, also play a key role in effective resource management. Finally, but most importantly, one must encourage use of any resource management facility that has been implemented on the system – no matter how good the facility is, it will not be effective in controlling system usage and resources unless it is actually used.

Below is a list of the key features that we noted as being essential for efficient resource management on Oscar:

- Simple easy to use front end commands.
  - allow a user to submit or delete requests, and to view the status of their requests, with minimum effort on their part
- Flexible scheduling / queueing system.
  - allow a user to submit any size request without being restricted to a number of fixed size requests, or without being restricted to running large requests at certain times of the day or week
  - minimum system administration
- Fair sharing of system resources between groups.
  - accounting system based on shares per group
  - prevent one user or group from hogging system resources
  - means of redistributing shares when the system is being under utilised
- Efficient use of system resources.
  - minimise number of cpu cycles that are wasted
  - minimise cache-thrash and swapping
  - maximise the throughput of jobs

Building blocks

From the outset it was decided to employ SGI's new resource management facility, *Miser*, as the features offered by Miser would enable us to more efficiently manage the system work load, without having to resort to partitioning the system. It was also decided to use Cray's workload management software *NQE*, as this would allow us to take full advantage of the batch queueing facilities of NQE, whilst at the same time utilise the resource management and scheduling offered by Miser. IRIX system accounting[4] would be employed to obtain information about the resources utilised by each users requests on a daily basis.

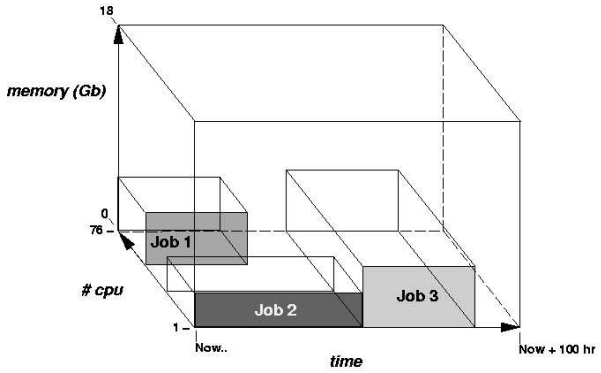

Figure 2: The Miser scheduling window

Miser

Miser is a resource management facility that schedules requests with known time and space requirements. Miser manages a time / space pool, consisting of a number of cpus, and a given amount of memory, from which it can allocate resources to run requests for a defined period of time, as shown in Figure 2:

Given a request and its resources (number of cpus, maximum amount of memory and total wall clock time), Miser will search through the time / space pool that it manages, until it finds an allocation that first fits the request. The request is then scheduled to run with a given start and end time, and the resources that have been allotted to it during this time. These resources are *guaranteed* by the kernel during the request's scheduled run time. Therefore, when the request's scheduled start time is reached, it will run without pre-emption until its scheduled end time. As such, the request should run more quickly, and have a predictable execution time, as it will not have to compete for system resources as it would do if it were run in the normal timeshare scheduling class.

It is important to note that the schedule allocated to a request by Miser is *non-conflicting*. At no point will Miser over-subscribe the resources that it manages. Therefore any request that exceeds the maximum amount of memory, or the total wall clock time requested when scheduled by Miser, will be terminated. On the other hand, any resources not being utilised by a Miser request will be made available for use by any other process on the system, but can be reclaimed back by the Miser request as necessary.

NQE

Cray's Network Queueing Environment (NQE) is primarily a workload management facility that provides batch scheduling and interactive load balancing across a heterogeneous network of Unix machines. Taking advantage of the Network Queueing System (NQS), implemented within NQE, it is also possible to setup a standalone batch queueing system that runs on only one machine. The advantage of Cray's NQE, is that it has also been configured to use the IRIX Miser scheduler as one of it's scheduling options. This therefore provides a means of setting up a number of batch queues on *Oscar*, to which a number of restrictions may be applied to control the number and size of requests running on the system as a whole, or by any one user or group, at any one time, whilst still taking full advantage of the resource management and scheduling features offered by Miser.

Accounting

IRIX system accounting provides a set of utilities that may be used to log certain types of system activity. Of particular use, is the ability to log process activity on the system. This enables us to monitor the number of programs that a particular user has run in any twenty four hour period, as well as giving us information as to the resources that these programs have used. Using this information it is possible to implement some form of credit / charging system, that can be used to bill individual groups based on the total amount of resources each user within that group has utilised in any one day.

3 Implementation of the batch queueing system

The resource management facility that was finally implemented on *Oscar* comprises a batch queueing / scheduling system and a credit / charging system to try to ensure effective management of system load and resources, as well as fair share of resources between the individual groups involved.

Batch queueing / scheduling system

The batch queueing / scheduling system itself comprises three levels:

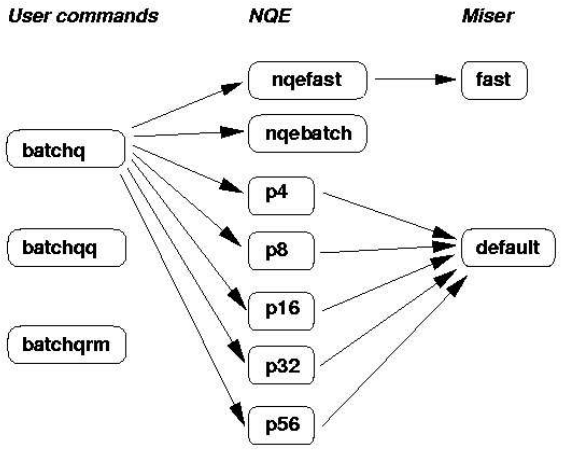

Figure 3: The OSC queueing system

Firstly, a set of user commands, *batchq*, *batchqrm* and *batchqq*, written in-house, allow the user to submit, delete or monitor the status of their request respectively. The second level comprises a number of NQE batch queues, where a request is either queued, passed on to Miser, or run. All bar one of these queues pass their requests to Miser for scheduling. The other, *nqebatch*, will run requests at low priority in the normal timeshare scheduling class. Finally, there are two Miser queues, where requests are scheduled to run in the high priority *batch critical* scheduling class, and are guaranteed their allotted resources during their scheduled run time.

The batchq command

The batchq command is used to submit a shell script as a batch request to one of the NQE batch queues on *Oscar*. The batchq command supports a number of options, but for any

script to be successfully submitted to one of the batch queues, the following options must be supplied as a minimum:

- the number of cpus to allocate to the request
- the maximum amount of memory, in Mbytes, that will be used by the request at any one point in time
- the total wall clock time, in minutes, for the request to run
- a shell script to be submitted as a batch request

During batch request submission, the batchq command firstly makes a decision as to which NQE batch queue to submit the request, according to the number of cpus requested and the batchq flags specified with the request.

A check is then made to ensure that the user's group currently has enough credits for the request to run, taking into account requests belonging to users in the same group that currently reside in one of the batch queues. If the user's group still has positive credits the batchq command will submit the request to the appropriate NQE batch queue for forwarding to Miser for scheduling. Otherwise the request is submitted to the *nqebatch* queue where it will be run at low priority in the normal timeshare scheduling class.

Finally, if the user requests more resources than the predefined maximum, currently 76 cpus, 18 Gbytes of memory, or 6,000 minutes of total wall clock time, the request is automatically rejected, and an appropriate error message displayed. These limits are configurable and can be changed to meet system demands or whenever the system configuration is changed.

The NQE batch queues

In all, seven NQE batch queues were setup on the system. Six of these queues, *p4*, *p8*, *p16*, *p32*, *p64* and *nqefast*, were configured to pass their requests to Miser for scheduling, the other, *nqebatch*, was configured to run requests at low priority in the normal timeshare scheduling class.

The purpose of the nqebatch batch queue is two fold. Firstly, it is for use by those groups who have insufficient credits to submit requests to one of the Miser queues. (It was decided that at no point would any group be restricted from using the system, even if they had used more than their fair share of resources.) Secondly, it serves to soak up any resources that are not being utilised by either of the Miser queues, along with interactive and system use, and therefore helps to minimise the number of cpu cycles that are wasted on the system.

To each NQE batch queue a number of limits have been applied to help restrict the number and size of requests running on the system at any one time. Firstly, each queue has an upper limit on the size of request that it will accept. But by far, the most important limit is the *Miser scheduling window*, that is applied to all NQE batch queues that forward requests to Miser. This time period controls how far into the future Miser will attempt to schedule a request. The request will only be moved from the batch queue to the Miser queue if Miser can schedule it to start running within the time period of the Miser scheduling window for that particular queue. If Miser is unable to schedule the request within this time period, the request will continue to remain queued in the NQE batch queue, until the next NQE scheduling pass.

More than one NQE batch queue can be configured to forward requests to the same Miser queue. On *Oscar*, five of the NQE batch queues, p4, p8, p16, p32 and p64, forward their

requests to the Miser queue *default*, whilst only one NQE batch queue, nqefast, has been configured to forward requests to the Miser queue *fast*.

The Miser scheduling window for each of these NQE batch queues has, at present, been defined so that the queues that forward requests that require larger amounts of resources are given a larger scheduling window compared to those that forward requests requiring smaller amounts of resources. This has the effect of biasing scheduling towards those requests that require more resources, as they are given a larger time period in which to be scheduled. (This may not, however, be the best approach and we are performing experiments to try to find an optimal set of parameters which will be described in section 4.)

Limits that restrict the number of requests that may be run, or the number of processors allocated to all requests in any one queue at any one point in time, are used only to control the number and size of requests running in the nqebatch batch queue, as Miser itself controls the number and size of requests running in either of the two Miser queues. An additional limit to control the number of requests run by one user at any one time was also applied to the NQE batch queues, as this hogging of resources by one user, even if their group had sufficient credits, was found to be a source of irritation amongst other users on the system.

Miser resource queues

Two Miser resource queues, *default* and *fast*, were set up on *Oscar*. Each queue has its own set of resources from which Miser can schedule requests to run. All requests are submitted to the appropriate NQE batch queue for forwarding to the default Miser queue, unless the user has specified that the request be submitted to the nqefast queue, for forwarding to the fast Miser queue, at the time of submission.

The fast queue has a limited number of resources, 4 cpus and 1 Gbyte of memory, with which to schedule requests. This queue is primarily for those users who wish to run small, short jobs, for example, when debugging code. A time limit of 120 minutes per request, is enforced by the batchq command at the time of submission.

The default queue, where the majority of requests are run under Miser, has 76 cpus and 18 Gbytes of memory with which to schedule requests. Requests are passed from one of the NQE batch queues. The size of the request, and the time in which Miser has to schedule the request being dependant upon the queue from which the request was forwarded, and is used to control the mix of requests being passed to Miser.

Credit / charging system

To try to ensure fair share of system resources between the individual groups involved with *Oscar*, a credit / charging system was implemented, where each group is given a certain credit allocation proportional to its share holding on the system. One credit is charged for each minute of cpu time used per processor. An additional charge is levied for excessive memory utilisation.

To each group's credit allocation is added a predefined daily income (proportional to its share-holding), and is subtracted a daily charge based on the resources that have been used in the previous twenty four hours. Upper and lower bounds constrain each group's credit limit, so that no group can accumulate credits indefinitely, nor can any group go into an arbitrarily large deficit. These bounds aim to try to encourage those users who are not using the system to their full potential to do so, "use it or you lose it", yet at the same time do not deter those users that are keenly using the system, by letting them incur a massive credit debt. Any group that has a negative number of credits is not banned from using the system, but is prohibited from submitting jobs to one of the favourable Miser queues.

The ability for one group to transfer some of its credits to that of another group, has also been implemented. This allows those groups which have a negative number of credits to barter for resources with those groups which are currently not actively using the system, and therefore help to minimise the number of cpu cycles that are wasted.

Process activity data is logged by the kernel every time a process terminates. A report of the system usage for the previous twenty four hours is generated on a daily basis. From this report, the resources utilised by each user and then each group is calculated, and a charge made according to the following formula:

$$\text{Charge} = \text{Max}\left(\text{cpu minutes}, \frac{1}{256} \times \text{MB minutes}\right)$$

A group is only charged for the amount of cpu time that it has utilised, unless it has utilised more memory than is available per cpu on the system. In which case the group is also charged for the additional memory that it has utilised.

4 Performance Optimisation

The queueing system in place has a number of configurable parameters which, up to now, we have set by trial and error to establish an acceptable throughput of jobs and mean delay time for a job getting scheduled. The most important parameters are the settings of the Miser scheduling window for each queue, which controls how far into the future a job may be scheduled to start.

So this presents us with an optimisation problem: what values to choose for these parameters to extract the best performance from the scheduling system. We shall need to define some formal measure of performance to use as our guide. As *Oscar* is over-subscribed we might consider the maximum attainable processor utilisation to be a reasonable criterion. However to concentrate on this to the exclusion of everything else might lead us towards a system with an unacceptable mean scheduling delay. We also have to consider whether the aforementioned mean scheduling delay should be weighted according to job size.

We have decided to tackle this problem by developing a simulation of the queueing system – a simple object-oriented program. This enables us to perform experiments with differing densities of job traffic, different settings for the queue parameters, and a variety of evaluation criteria. An alternative strategy, which we are yet to investigate, would be to apply queueing theory[5].

Simulating the batch queueing system

The simulation program: `OSCQ`, which has been developed in an object-oriented style, is constructed from objects: *Users*, *Batchq*, *NQE*, and *Miser*, which encapsulate the data and behaviour of the various components of the system. Within each object there are various configurable parameters, corresponding to actual system parameters. However, note that it would be infeasible for us to implement the full functionality of NQE and Miser in our simulation – we abstract away certain details that are assumed to be irrelevant to our investigation. For instance, we are, for now, ignoring that fact that users may decide to kill a job at any point using the `batchqrm` command. We also assume that a job will run for the exact amount of time that was requested, whereas, in practice, it may finish much earlier, either because the user requested more cpu time than was necessary, or because of program failure. Whether or not these assumptions affect the result of our optimisation work will be investigated later.

A single run of the OSCQ program takes, as input, a list of time-stamped job requests: each of the form

$$(\text{Submission time}, \text{Processor}_{max}, \text{Memory}_{max}, \text{Time}_{max})$$

plus a list of values for system parameters: $(p_1, p_2, ..., p_n)$. It then works out, through simulation over discrete time-steps, the point at which each job would actually get scheduled. This data is then processed to derive performance statistics.

The main performance statistics that we calculate at present, for a particular run of OSCQ, are as follows:

Processor utilisation: U – percentage of time that processors allocated to Miser are kept busy for the duration of the run.

Mean delay: W – average time between submission of a job and actual scheduling.

Weighted mean delay: WW – average of $\frac{\text{time between job submission and scheduling}}{\text{total number of cpu minutes requested}}$

Overload factor: O – the maximum number of jobs simultaneously in the system: either queued or running.

Simulating job traffic

In order to perform optimisation experiments with the OSCQ program we need a means of simulating job traffic, at varying rates of throughput. There needs to be a certain random element in this to ensure that we are not purely optimising the system for a particular set of test data. The strategy we are using at present is to analyse the distribution of actual jobs on our system with respect to parameters $(\text{Processor}_{max}, \text{Memory}_{max}, \text{Time}_{max})$, and then to generate a random sequence of jobs fitting such a pattern. A random submission time is attached to each job, assuming, for now an even distribution as to when jobs get submitted. In practice the distribution is uneven, as less jobs are submitted during the night than during the day. We are also ignoring any 'feedback-factor', the extent to which people refrain from submitting jobs into an over-subscribed system. These factors will be investigated later.

This leads us to a traffic generator program, TRAFFIC, which takes two arguments:

- NJOBS – the total number of jobs to generate
- TIME – the time interval in which all the jobs are to be submitted.

Different traffic densities may thus be simulated by altering the ratio of NJOBS to TIME.

Optimisation of parameters

It is planned to use the technique of *genetic algorithms* to search for an optimal set of parameters $(P_1, ..., P_n)$ for the queueing system. But, at present, we are still in the early stages of this research.

The general idea is as follows. We need to define an evaluation program, EVAL, which will take a particular set of system parameters, run a number of simulation experiments, using random traffic data, and then return a single number G, to represent how good those parameters are for running the system. It is important to ensure that any pathological behaviour that we can envisage would be reflected by a poor evaluation score.

Once a reliable evaluation procedure has been established, we can perform a standard genetic optimisation experiment. We shall start with a pool of widely varying parameter sets, or *gene-sequences*, $(S_1, S_2, ..., S_m)$, where $S_i = (P_1^i, ..., P_n^i)$ and then evaluate the performance of each S_i using `EVAL`. A "survival of the fittest" replacement strategy will then be employed. There are many variations of this but, in the most simple form, on each iteration one performs the following steps:

1. Select the two strongest gene-sequences: S_i, S_j
2. Mate them using the technique of crossover:

$$S_i' = (P_1^i, ..., P_k^i, P_{k+1}^j, ..., P_n^j)$$

$$S_j' = (P_1^j, ..., P_k^j, P_{k+1}^i, ..., P_n^i)$$

   for some random k, to produce two fresh offspring.
3. Replace the two weakest gene sequences from the original pool with S_i' and S_j'.

This particular strategy may need to be extended to avoid convergence to some local maximum which does not represent an optimal solution to the overall problem. There may also be some advantage in looking at more complicated reproduction techniques.

5 Current Status and Future Plans

We have experienced a number of minor problems which have interfered with the smooth running of our queueing system in its early days. Mainly these have related to the unclean interface between Miser and NQE. Despite these problems, our users have heartily embraced the system because of the huge benefits offered by dedicated resource allocation. We hope to enhance the overall performance of the system, in time, as a result of our ongoing optimisation project.

In the future SGI will be switching support from NQE to the alternative LSF queueing system, but we do not see any major impact to the way that we are running *Oscar* as a result of this.

One significant problem with Miser, as it stands, is that we have limited control over when a job gets scheduled, which virtually rules out interactive work on a busy system. We are planning to invest in special-purpose visualisation hardware to be incorporated into *Oscar*, and it is hard to see how this could be used effectively within the existing batch framework. We require the ability to set fixed starting times for our jobs, assuming that the resources we need are available.

References

[1] *Origin Servers Technical Report*, Silicon Graphics Inc

[2] *IRIX Admin: System Configuration and Operation*, Silicon Graphics, Inc.

[3] *Introducing NQE*, Cray Research, Inc.

[4] *IRIX Admin: Backup, Security and Accounting*, Silicon Graphics, Inc.

[5] A Tanenbaum, *Computer Networks, 2nd Edition*, Prentice-Hall 1988