

# Time-Critical Scheduling on a Well Utilised HPC System at ECMWF using Loadleveler with Resource Reservation

Graham Holt  
ECMWF  
Shinfield Park  
Shinfield Road  
Reading  
Berkshire  
UK  
RG2 9AX

*ECMWF (European Centre for Medium-Range Weather Forecasts) was founded in 1973 and is funded by 25 European Countries (18 original participating Member States and 7 cooperating Member States) where Medium-Range Weather Forecasts concentrate on the period 4 – 10 days ahead. The European Weather Centre, as it is more commonly known, is located 35 miles west of London, England on the outskirts of a town called Reading. See <http://www.ecmwf.int/about/overview/> for more information.*

email: [G.Holt@ecmwf.int](mailto:G.Holt@ecmwf.int)

## **Abstract**

*This article is written in the context of running a suite of time-critical operational numerical weather prediction batch jobs along with a substantial number of research batch jobs on a large IBM Cluster 1600 system. The batch subsystem used is IBM's LoadLeveler incorporating a little known feature called Resource Reservation.*

*The article describes how the mixture of operational and research parallel batch jobs are scheduled to run on the 117 nodes provided and how Resource Reservation for operational jobs is performed without reference to job class. Where research parallel batch jobs are jobs requesting more than 1 CPU and must run consistently to ensure resources are released predictably. Note - information is given explaining how consistent runtimes are achieved.*

## **1. Background information.**

Before 2001, ECMWF had no experience of Loadleveler, having previously used systems from CDC, CRAY (NQE) and Fujitsu (NQS). So ECMWF's experience of Loadleveler is limited to the needs of the system described, and the scheduling strategy devised in early 2002 was kept simple to minimise the learning curve/time. More recently additional IBM Server systems again using Loadleveler have been installed, but experience has shown there is little common ground between the philosophy of scheduling batch and interactive work on Servers and the philosophy of scheduling high-performance parallel batch jobs on a Cluster. So my Loadleveler/scheduling experience is therefore fine-tuned in a blinkered way to the needs of well balanced, highly parallelised batch jobs on a large Cluster to ensure good performance and consistent runtimes. Importantly the over-subscription of processes per node is not allowed, shared memory use is only permitted by up to 4 jobs per node, maximum 8 processes (as long as the total real memory requested by all the jobs does not exceed the total real memory available), and memory paging when detected is reported as very undesirable, even more so if job performance (CPU utilisation) appears to be compromised.

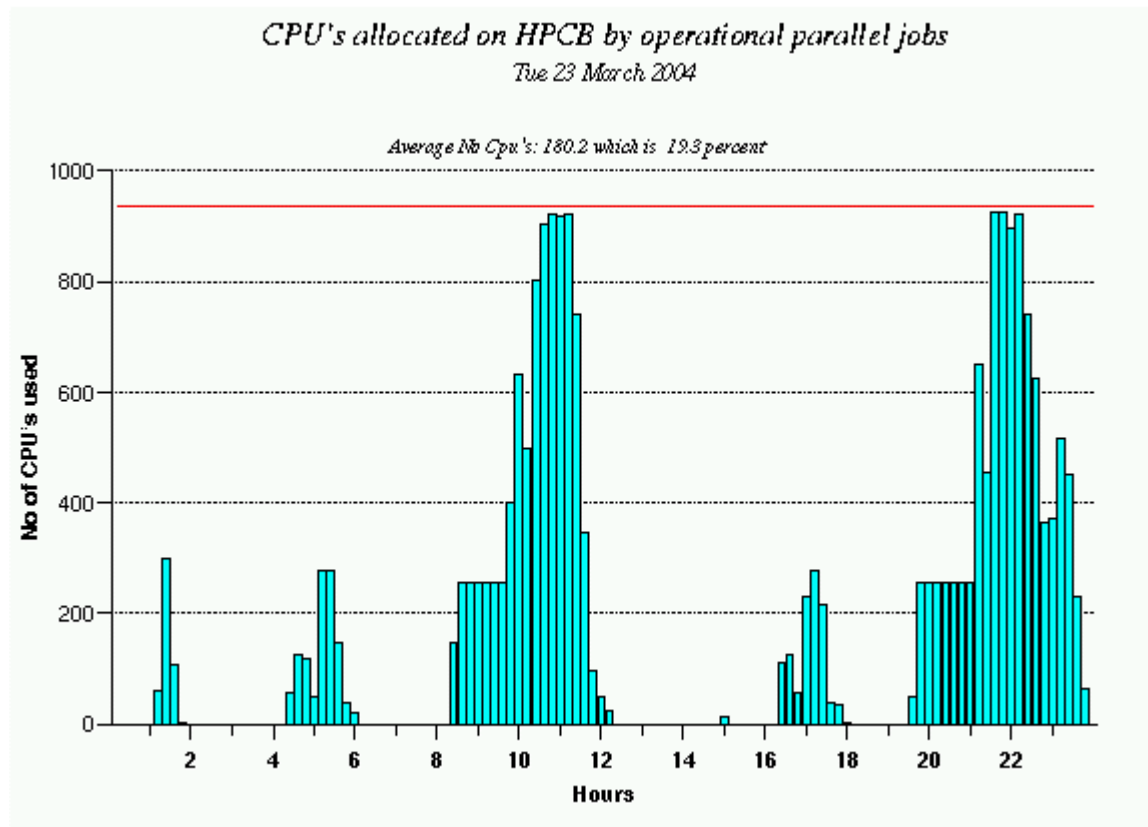
In 2002 the Loadleveler feature Resource Reservation was not known to ECMWF. At that time when operational jobs were about to be run nodes were 'reserved' by draining 'user' batch job classes so that no new 'user' jobs could start. But this manual, time consuming and often complicated method was frequently found to be wasteful, with nodes being left idle unnecessarily. Then in 2003 a decision was taken to introduce a tighter operational schedule and as a result we needed to develop an automated resource based scheduling scheme that would also overcome the known weaknesses in the manual system in use.

It was clear even in 2002 that predictable runtimes were essential for backfill to maximise node utilisation and for predictable node release. So it was agreed the scheduling scheme like backfill should be knowledge based and use

predicted wall\_clock\_limits derived from historical run-time data. A lot of work had already gone into creating tools and displays that enabled runtime data to be captured, visualised, and made available to enhance job selection and empower backfill. Importantly, having no previous understanding of IBM backfill, tools had already been created to monitor the results of backfill so that the way it worked could be studied and understood. So plans were made to enhance these displays for use during operational periods. But most importantly a dynamic reservation based scheduling scheme was sought, a scheme independent of physical nodes and physical classes. For this IBM suggested using Resource Reservation.

## 2. Overview.

ECMWF runs many types of operational forecasts on a daily, weekly and monthly basis, and hundreds of thousands of products are sent (disseminated) to the Member States each day. Twice a day for about 90 minutes all 117 nodes (936 CPUs) are reserved for and used by operational batch jobs (see Figure 1 below)



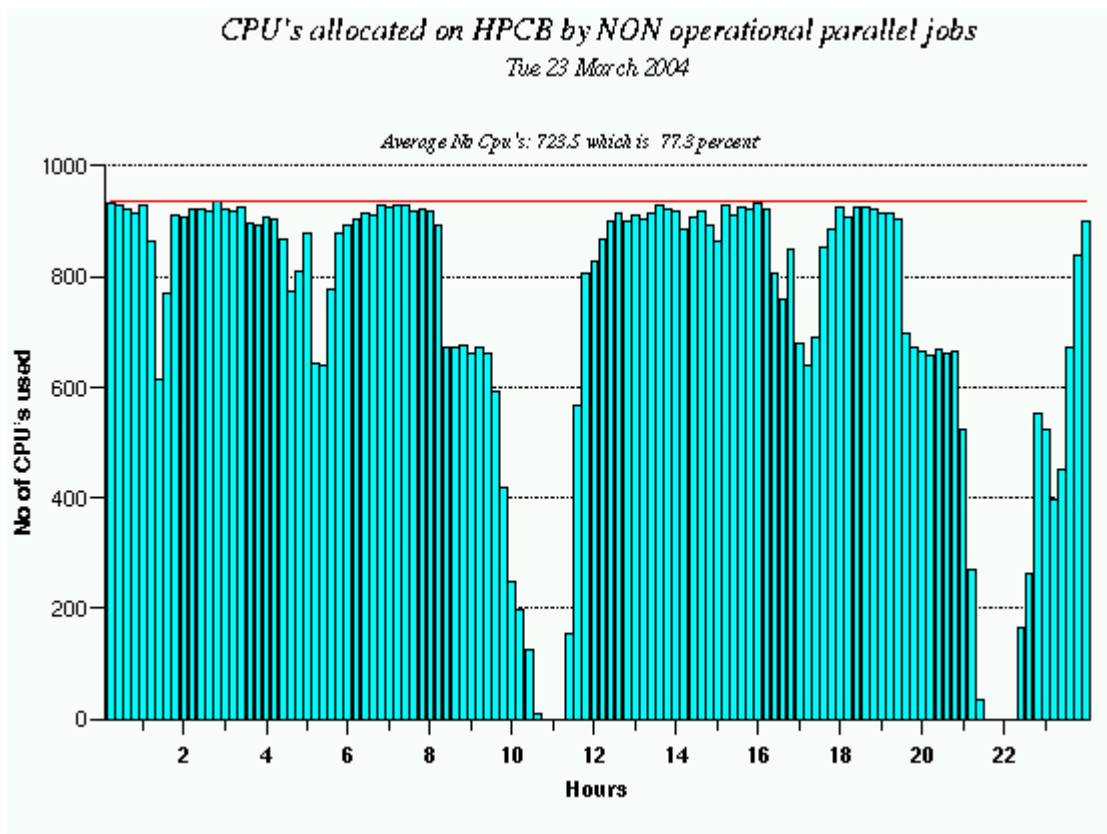
**Figure 1 - the average number of CPU's allocated to parallel operational jobs, plotted over 12 minute intervals, where for parallel jobs 936 CPUs (the red line) is the maximum possible.**

Please note – when all 936 CPUs are reserved for operational jobs, some CPUs become unallocated for a few seconds as jobs complete and new jobs start, and at times due to the job-mix at least 1 node (8 CPUs) may not be used for some minutes. So the average maximum use of around 920 CPUs out of 936 is seen as a very good achievement.

However when operational batch jobs are not being run or do not require all of the 'parallel' nodes, every attempt is made to fully utilise 'resources' by running research user's batch jobs

## 3. A mixture of parallel operational and research batch jobs.

Please note there is little difference between the computational needs and performance characteristics of research batch jobs and their operational equivalent jobs. The codes used are almost identical. The big difference is operational suites of jobs are run usually twice or four times a day and each time use as input the most recent world-wide observations acquired in the preceding few hours. But research experiments differ in that they use historical data (not real-time data acquisition), have no need to create end-user products and over a period of many days submit 28 data assimilations and forecasts, using alternately 00z and 12z data over a 14 date period. Normally there are from 10 to 15 research experiments running simultaneously and these plus other smaller jobs easily utilise the 117 nodes available (see Figure 2 below) whenever resources are not being used operationally. Please note – 00z data is collected globally and simultaneously at 00z GMT, likewise 12z data at 12z GMT.



**Figure 2 - the average number of CPUs allocated to non-operational jobs, before Resource Reservation was introduced, plotted over 12-minute intervals, where 936 CPUs (the red line) is the maximum possible.**

Please note in Figure 2 above at 02:48, a 12-minute period when all 936 CPUs are allocated.

#### **4. Scheduling requirements.**

One additional function of pre-operational testing is to detect research jobs that do not perform well or do not exhibit consistent runtimes in keeping with the existing operational schedule. Quite simply, jobs (new code) from research experiments cannot progress beyond the research stage to become operational if they do not perform well enough. So a lot of effort has been put into ensuring all parallel jobs have consistent runtimes, and as a result each job confirms by running in the given time that there are no I/O bottlenecks, the GPFS filesystems used are performing consistently, the high-performance internal switch network is performing consistently, that the jobs have not been slowed by paging to local disk and lastly that the jobs (the new code) perform as expected too.

As a result, in a research experiment of jobs run 28 times, at least 95% of the forecast jobs are not expected to vary by more than  $\pm 1\%$ , with the preceding sets of 28 data assimilation jobs varying by no more than  $\pm 3\%$ . The big bonus being, once 3 or 4 sets of jobs for an experiment have completed, the subsequent 24 sets of jobs become health checks (each a diagnostic) for good system performance. It is true when a job has run less than twice it has an unknown runtime (shown on displays as a wall\_clock\_limit of \*1-day) but other jobs with similar job-names (for example ifstraj\_uptraj\_0) are likely to have a known run-time, giving an indication when new jobs might complete.

Importantly the golden rule before submitting an experiment or individual parallel job is – if the data needed by a parallel job is not on a GPFS filesystem, a serial job must first be run to obtain the initial data, and the data obtained must be written to a high-performance GPFS filesystem. In this way all parallel jobs in the Cluster can be scheduled independent of all other systems.

If at any time a parallel job runs for longer than expected or the job is flagged as idle, the job is displayed in red on an operator display and immediately investigation begins. What has to be determined is – is the problem a function of the job, the environment or the system?

#### **5. The mission - to keep the system fully utilised yet run operational work to a tight schedule.**

##### **5.1 Fully utilised**

Keeping the system lightly loaded to make it easy to run operational work on time is not our style (is for wimps). Users in the Research Department have always been able to expand their experimentation to fully utilise the system, and giving them the service they want by fully utilising the system is very rewarding.

## **5.2 Keeping to operational schedules.**

The plans made in 2003 to run the operational suites to a more demanding schedule made the introduction of an automatic and class-independent resource reservation system essential. The previous ‘on demand’ scheduling scheme could not guarantee consistent start-times, the variability often exceeded the 15 minute requirement, and as a result there was often insufficient recovery time should jobs fail and need to be rerun.

## **6. The main objective.**

The main objective of the reservation system was then defined like this: Resource reservation is needed to ensure that the day-to-day variation in the end-times of operational forecasts and the generation of products does not exceed 15 minutes compared with the predicted optimum time, whilst keeping node occupancy close to 96% (which is what had been achieved beforehand).

Fortunately the basic building blocks were all in place based on the predicted run-time for most jobs, the standard design of most experiments and the repetitive consistently running job mix, giving the ability to accurately assess node release and node availability times.

### **6.1 The complexity of operational suites should not be underestimated.**

Until now I have trivialised operational suites at ECMWF by talking only about the more substantial time-critical computational jobs. The reality is each suite contains 1000's of jobs that run on multiple systems, have complex inter-dependencies, multiple-event triggers, time triggers and late flags. Amongst other things, operational jobs acquire observations (data), analyse and check the validity of this data, compute the relevant initial datasets used by the forecast, execute the forecasts, save data at many stages throughout the process, create end-user products, verify the results, archive the results, plot data, send products to users with many operations taking place in parallel. One concern with such a complex set-up, was tighter schedules would lead to new bottlenecks and a complete loss of flexibility, which would reduce efficiency and system utilisation. So the brief was altered subtly to request an automatic system that used the 15 minutes of flexibility if this increased system utilisation.

## **7. Resource Reservation.**

### **7.1 Defining the need for resources and benefits of Resource Reservation.**

The CPU allocation averages shown previously in Figure 1 indicate that most of the time none of the Cluster resources are used by operational jobs, and show during the main periods (08:20 to 11:40 and 19:45 to 23:35) the need for the resources provided to increase in 3 stages from 0% to 30%, from 30% to 60%, and finally from 60% to 100% for about 90 minutes. But what Figure 1 does not show are the 2 small periods when CPU use drops for just a few minutes from 30% to 0% before rising to 60%, and later from 60% down to 30% before rising up to 100%. If resources are not reserved or classes drained, user jobs flood the system.

Past experience showed it is possible, but by no means easy, to drain classes on some nodes to limit the use of resources by non-operational jobs to 40%, whilst the operational use dips briefly from 60% to 30% before rising to 100%. By comparison dynamic reservation independent of class using Resource Reservation provides a much needed simplicity. One might even say Resource Reservation brings elegance to the solution.

And by the same token with one operation, as soon as all operational jobs to be run are submitted, and because operational jobs are selected first, by virtue of highest SYSPRIO, it is possible to cancel (reset) reservation, make all non-operational jobs candidates for selection before the final operational jobs have started and allow backfill to take place.

### **7.2 Describing Resource Reservation.**

Resource Reservation for the timely running of operational jobs combined with optimum system utilisation has as a core function the ability to request in advance a reduction to the resources available to non-operational jobs, irrespective of the class the jobs are in, whilst taking into account

- . the requirements of operational jobs (expected start-time, CPUs needed x elapsed time),
- . the release of resources (expected elapsed time remaining for research jobs in execution)
- . the estimated start-times (or delays if research jobs appear to end soon enough)

Additionally the scheme should take into account

- . the flexibility of the operational schedule

- . how close to or far behind the optimum schedule today's operational runs are
- . how consistent the historical runtime data is for the jobs that are running,
- . that it is possible to force some jobs to write checkpoint files then kill them without loss.

Finally the scheme should release resources for use by non-operational jobs whenever possible to optimise the use of all resources as shown by Figure 3 below.

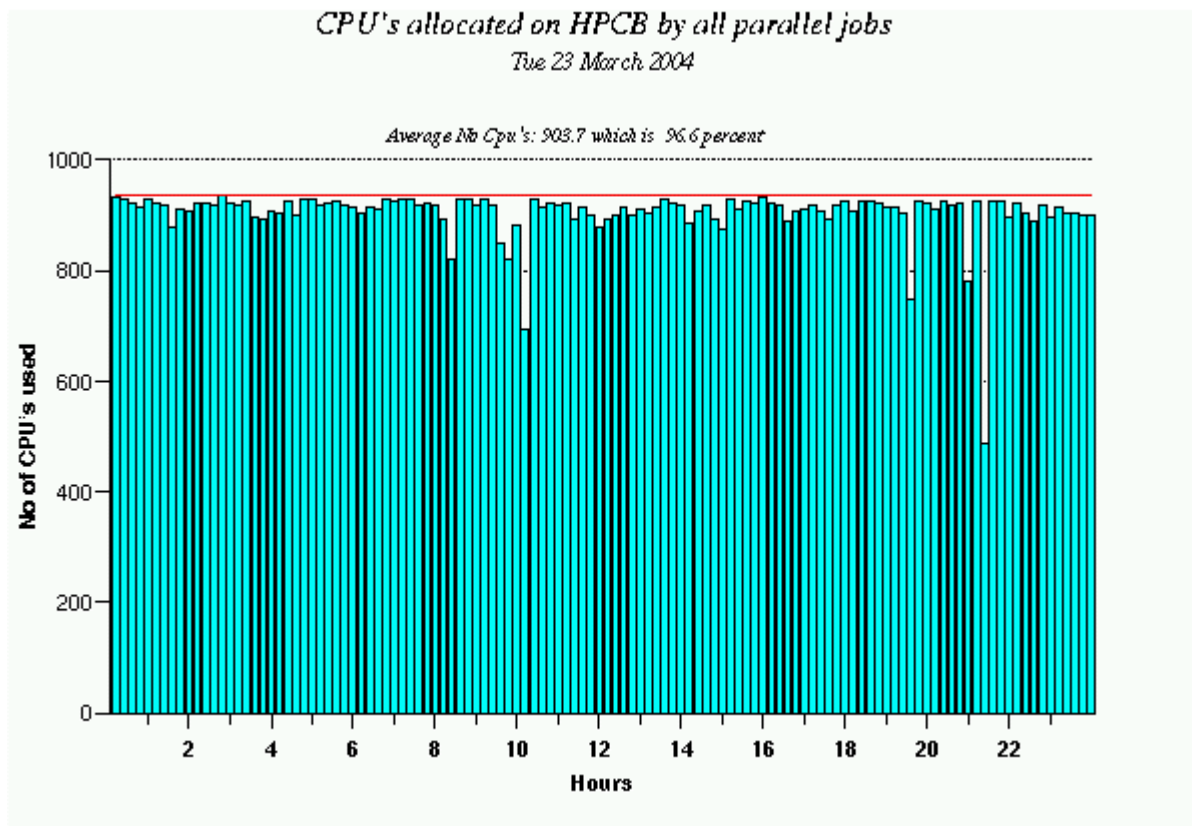


Figure 3 - the allocation of CPU's to all jobs before Resource Reservation was introduced, plotted over 12-minute intervals, where the red line (936 CPUs) is the maximum number that parallel jobs can use.

## 8. Summary so far.

The key to everything is predictability. ECMWF's business relies on the timely production of weather forecast products (our weather predictions) so not surprisingly ECMWF has a strong desire for all jobs to run predictably. Otherwise how do you keep to a schedule. And once the causes of variable job runtimes are eliminated (as they have been) and jobs run with little run-time variation, the data available enables

- . backfill to work most effectively.
- . the runtime and end-time of user batch jobs to be monitored.
- . most batch jobs to act as diagnostic checks on system performance as well as on job performance.
- . resources to be reserved sufficiently in advance to ensure operational jobs start optimally.

## 9. Defining the core elements required for effective Resource Reservation.

Analysis of the associated issues have identified 4 main elements, which are listed below so that they may be checked and ticked off. In doing so one can be sure all problems have been identified and solutions put in place. It is true there is some duplication of what has gone before but this is necessary. The 4 elements being

- A. The need to know in advance what resources are required by operational jobs and the time at which these jobs should start.
- B. The need to ensure that non-operational jobs in execution can be guaranteed to complete and release the resources in a predictable way (there is little point doing this if, in order to achieve it, many jobs have to be cancelled prematurely).

C. The need to have processes that take the information about operational jobs and running jobs and use this along with other information to dynamically reserve resources, the purpose being to restrict or stop non-operational jobs from starting, then release resources.

D. The need to be able to visualise what is happening and verify that it is working as designed.

## 10. Expanding on the core elements for effective Resource Reservation.

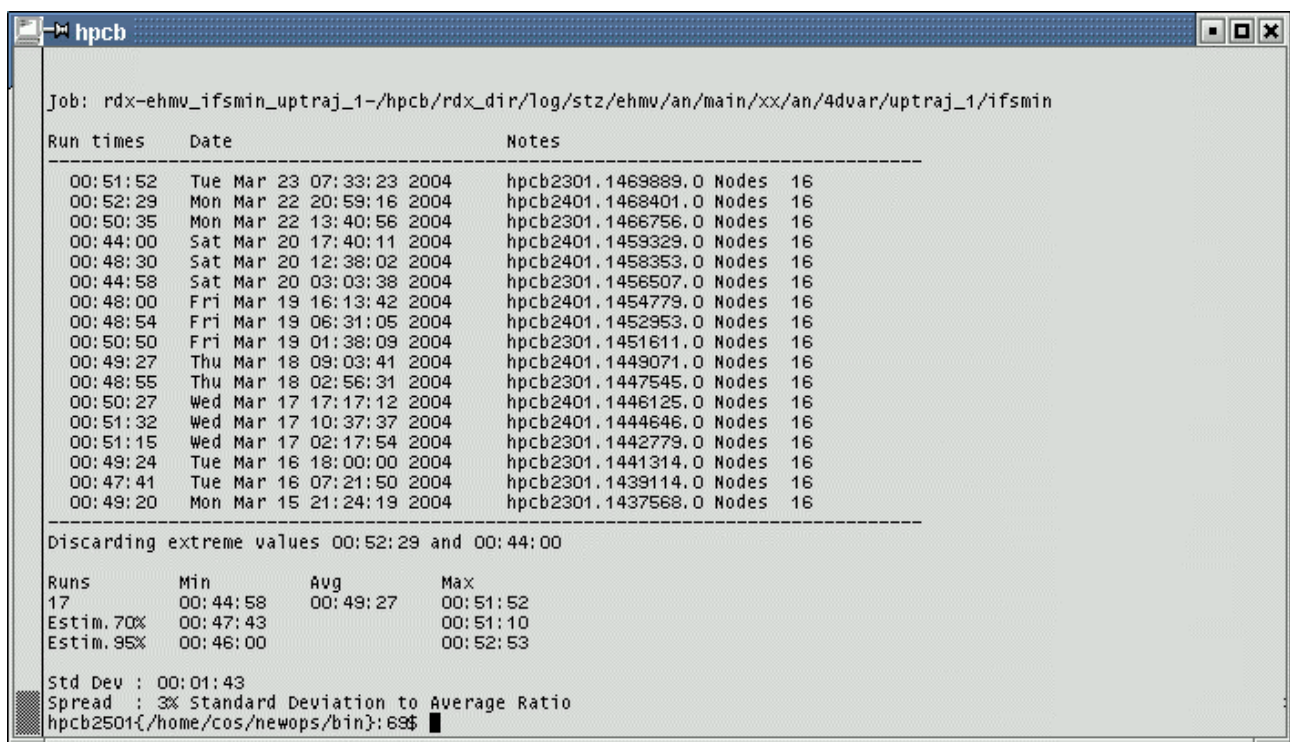
Taking the heading from 9. (and covering A. and B. very quickly) we have:-

**A. The need to know in advance what resources are required by operational jobs and the time at which these jobs should start.**

The requirements of operational jobs, in terms of resources, start time, elapsed time and dependencies on other factors are well known. Nothing more needs to be added.

**B. The need to ensure that non-operational jobs in execution can be guaranteed to complete and release the resources in a predictable way.**

As mentioned before, predictability is paramount. There is no point reserving resources if job end-times are unreliable. Good utilisation is compatible with tight schedules only if you have a predictable set of non-operational jobs. The key elements of ECMWF's success are the independence of parallel jobs, consistent job performance and consistent system performance. And to illustrate the point Figure 4, which follows, shows historical run-time data for an assimilation job and the predicted Min and Max runtimes. Where the predicted 95% Max value is used for scheduling and backfill purposes. The job selected does not quite have the quoted runtime range of  $\pm 3\%$  and this is commented on later.



**Figure 4 - historical run-time data and predicted run-time estimate (95%/Max) for job ehmv\_ifsmin\_uptraj\_1**

Figure 4 shows 17 runtimes. Discarding the shortest and longest, the average runtime (under Avg) is shown to be 00:49:27. The expected  $\pm 3\%$  would give Min 00:47:59 and Max 00:50:55. Sadly 6 runtimes fall outside this range. The larger than expected spread of 00:44:58 to 00:51:52 means the predicted 95% Max runtime of 00:52:53 exceeds by 24 seconds the known longest run of 00:52:29, and exceeds the average by almost 3.5 minutes. But this is no disaster. It means successive jobs will end about 3.5 minutes before the predicted run-time and as a result 16 nodes out of 117 may be idle for 3.5 minutes longer than expected before operational jobs run. Less-than ideal but never the less still very acceptable.

What is clear however is the basic requirements exist for the introduction of a resource reservation system. We know in advance the resources needed by operational jobs and the time these jobs should start. We have confidence that the non-operational jobs have well predicted run-times and will finish as required/expected.



C. The need to have processes that take the information about operational jobs and running jobs and use this along with other information to dynamically reserve resources, the purpose being to restrict or stop non-operational jobs from starting, then release resources.

To reserve resources IBM suggested ECMWF use a little-used Loadleveler feature called “floating resources”, where “floating resources” are in effect cluster-wide licenses to use CPUs. The floating resources we use are defined as “FloatingCpus”, the unit of which is a physical CPU. These are requested by non-operational jobs in the same way as other resources, such as ConsumableCpus and ConsumableMemory via the “resources” LoadLeveler directive. e.g.

```
# @ resources = ConsumableCpus(4) ConsumableMemory(3600Mb) FloatingCpus(4)
```

And as it is imperative that all non-operational jobs contain such a directive and request the correct number of FloatingCpus all jobs are passed through a “job filter” and the job filter inserts the request. Then by reducing the number available, it is possible to restrict all non-operational jobs to a limited number of CPUs, and ensure that only operational jobs, which by design do not request floating resources, can use the remaining CPUs.

With 117 nodes, each node an 8-way SMP system, making 936 CPUs in total, the following directive in LoadLeveler’s configuration file:

```
FLOATING_RESOURCES = FloatingCpus(936)
```

means that outside operational periods we make all 936 CPUs available for running non-operational work.

Then when it becomes necessary to reduce the non-operational workload, to reserve CPU’s for operational jobs, the value of cluster-wide FloatingCpus limit is reduced in LoadLeveler’s configuration file, and by signalling the relevant LoadLeveler daemons the change is introduced. As non-operational jobs finish and release nodes, the number of FloatingCpus in use is reduced but no new non-operational jobs will start until the number of FloatingCpus in use becomes less than the value of FloatingCpus set in the LoadLeveler configuration file. Of course, as this is a reservation scheme, the value is reduced some time before the operational jobs are submitted, taking into account the time needed for user jobs to end (which will be covered later).

Ignoring the time in advance calculation for now, to reserve 256 CPUs for the first operational job, the value of FloatingCpus in the LoadLeveler configuration file is reduced from:

```
FLOATING_RESOURCES = FloatingCpus(936)
```

to

```
FLOATING_RESOURCES = FloatingCpus(680)
```

and once 256 CPUs become available, they will remain unused until the operational job starts.

However, the skill is in working out when to reduce the value of FloatingCpus. Too early and resources will lie idle, waiting for the operational jobs to be submitted. Too late and the resources will not be released in time. However as the operational schedule has a degree of flexibility a little late is better than a little early.

Clearly much depends on the number of nodes needed (to give time for a running job or running jobs to end) and taking into account the known flexibility in the running of operational jobs. To maximise the use of the system it would be beneficial if shorter non-operational jobs with the most consistent historical run-times were to be selected just before operational periods. Additionally if the operational resources required are needed in stages rather than all at once, a more flexible approach can be made.

ECMWF talked to IBM about this. IBM is proposing to implement a resource reservation scheme that, like LoadLeveler’s backfill scheduler, works by using a job’s wall\_clock\_limit. Sadly on a basic IBM system wall\_clock\_limit is rarely set accurately, and if it is the kill on wall\_clock\_limit exceeded is a rigid scheme that is not acceptable to ECMWF. So a local fix was needed.

## 11 The backfill problem.

### 11.1 First identify the problem.

IBM ask users to set the wall\_clock\_limit and then use the value given for backfill. Yet users are known to have a poor understanding of the length of time their jobs will take to run and even on the ‘best’ systems runtimes will vary. So expecting users to keep tabs on the spread in unrealistic. Then add to this the basic philosophy behind wall\_clock\_limit, which is to kill jobs which over-run, and it is no wonder that users never specify an “accurate” value. You do not have to be a rocket scientist to realise I have no faith in scheduling and backfill using user-specified wall\_clock\_limits. The system forces users to be inaccurate.

### 11.2 The ECMWF solution.

- Ask users to not specify a job wall\_clock\_limit.
- When each job completes, to collect data (including actual run-time) and store it in a database.
- To predict wall\_clock\_limits using the data collected.

If a job (same name, same user) is submitted more than twice, the job filter uses the historical run-time data to create a predicted run-time, see Figure 4 using the Estim.95%/Max value 00:52:53 for this purpose. Then add one day to this and set the wall\_clock\_limit to “1+00:52:53”. The time added ensures the job is not killed should it over-run its’ expected runtime and the 1+ is used because it is very easy to code. Additionally it is very easy to mask out in displays leaving the true predicted run-time for all to see. Please note - the backfill scheduler works by comparing time differences. So provided all wall\_clock\_limits use the 24-hour baseline, which they do, the backfill scheduler functions as designed. The 1+ is only an offset used to stop backfill from killing jobs, it can otherwise be ignored.

As pointed out earlier, job runtimes are quite consistent, and the predicted run-times are always a slight over-estimate using the “Estim.95%/ Max” value, so it is quite rare for jobs to over-run. It is much more common for non-operational jobs to end about 1 to 2 minutes ahead of the predicted run-time. But should a job run for more than 1 minute longer than the predicted run-time it will be highlighted in red on the operator display, to ensure investigation starts as soon as possible. An example of this comes later with Figure 7.

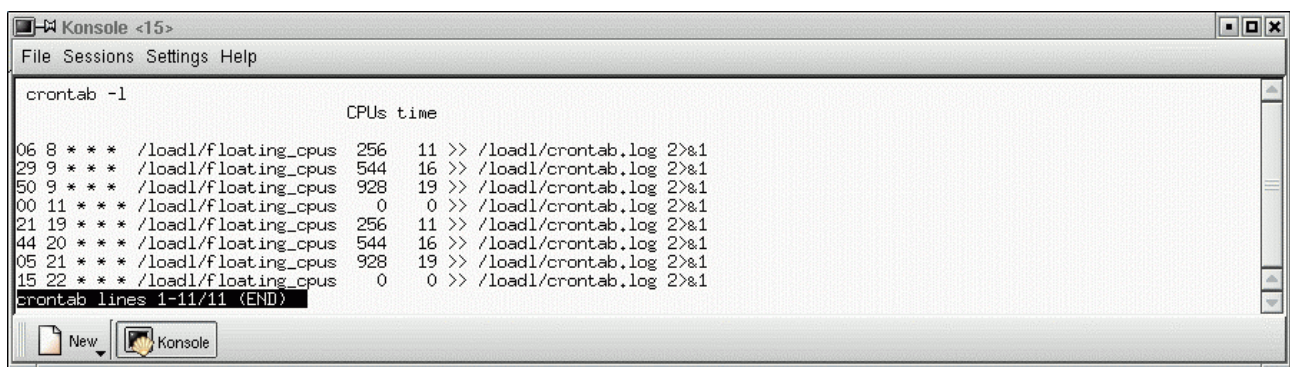
Up to now I have described all of the elements that enable a system to predict and control the reservation, use and release of CPUs (nodes). I have shown that the data exists to enable nodes to be filled and to ensure nodes are released in line with tight operational schedules. Theory is fine, but how is it done in practise?

## 12. When does the value of FloatingCpus get modified in the LoadLeveler configuration file?

Analysis of the workload is used to determine the average time for nodes to be released. And from this we have determined the time in advance (in minutes) for each reduction of FloatingCpus. The values chosen can be seen under the heading ‘time’ in Figure 6. A more sophisticated scheme could be used but this simple scheme works and has not yet been improved on.

## 13. How does the value of FloatingCpus get modified in the LoadLeveler configuration file?

A utility was written to modify the LoadLeveler configuration file “/loadl/config”. This utility “floating\_cpus” takes care of file locking to ensure that the file is not being manipulated by other utilities concurrently. At present “floating\_cpus” is executed via “cron” at the times shown in Figure 6. The call to run “floating\_cpus” comes with 2 parameters, the number of CPUs to be reserved and time in advance the new value is being applied.



	CPUs	time
06 8 * * *	/loadl/floating_cpus 256	11 >> /loadl/crontab.log 2>%1
29 9 * * *	/loadl/floating_cpus 544	16 >> /loadl/crontab.log 2>%1
50 9 * * *	/loadl/floating_cpus 928	19 >> /loadl/crontab.log 2>%1
00 11 * * *	/loadl/floating_cpus 0	0 >> /loadl/crontab.log 2>%1
21 19 * * *	/loadl/floating_cpus 256	11 >> /loadl/crontab.log 2>%1
44 20 * * *	/loadl/floating_cpus 544	16 >> /loadl/crontab.log 2>%1
05 21 * * *	/loadl/floating_cpus 928	19 >> /loadl/crontab.log 2>%1
15 22 * * *	/loadl/floating_cpus 0	0 >> /loadl/crontab.log 2>%1

Figure 6 - the crontab runtimes where the first parameter is CPUs reserved and the second value is time (minutes), that is minutes in advance of the operational need.

Taking the first cron entry from Figure 6

CPUs time

06 8 \* \* \* /loadl/floating\_cpus 256 11

At 08:06 the crontab job /loadl/floating\_cpus is run to reserve 256 CPUs. This is done by altering the value of FloatingCpus in the /loadl/config file, reducing the value from 936 to 680 with the knowledge that the 256 CPUs requested will be needed. The value of time (11 minutes) means the first operational job is expected to be submitted at 08:17.



In the future, the script ‘floating\_cpus’ will be executed by an event that is an integral part of the operational suite of jobs. Then if operational schedules are changed, the reservation times will automatically change too. However the “cron” mechanism will be retained as a safety net. If the start of operational activity is delayed significantly and the event-linked reservation of nodes does not run, a “cron” run a little later will ensure that resources are reserved so that when the operational activity eventually starts it is not be delayed further. Although potentially wasteful, on the few occasions when forecasts run very late it is essential that the resources needed are already available.

#### **14. When does the value of FloatingCpus get modified in the /loadl/config file?**

The utility “floating\_cpus” that sets the limit of FloatingCpus in the configuration file, runs ahead of the time reserving resources based on our experience of the normal average release of nodes. We plan to change this “rule of thumb” mechanism to one that analyses running jobs, and pre-selects the ‘right’ jobs. But currently this is not done.

It is obvious that schemes to alter job selection before operational periods will influence the use of nodes, the release of nodes and the optimum time in advance that resource reservation should be made. The opportunity for complexity is large. However we keep this simple by using average node release patterns and checking to see what actually happens. There is some waste, but to start with if there is less than a 15-minute variation to the runtimes of the operational suites we have achieved our primary aim. Slightly better node utilisation could be obtained but not a lot. Integrating Resource Reservation into the operational suite comes first.

#### **15. Summarising A. B. and C.**

The components of Resource Reservation are

- CPU resources called FloatingCpus defined in the /loadl/config file
- the job-filter ensures all non-operational jobs request FloatingCpus
- known requirements for all operational jobs
- historical run-time data for most user jobs
- A predicted run-time for most user jobs
- Confidence factors (see Figures 4 and later Figure 7 Std. Dev and Spread) in the predicted run-times.
- User job wall\_clock\_limits set to 1+ predicted run-times.
- Efficient backfill but with kill on wall\_clock\_limit disabled.
- A utility called floating\_cpus that sets new values of FloatingCpus in the config file
- No need to drain or resume job class globally or node by node.
- A utility that gives information about the average frequency of released nodes
- cron jobs that ensure the value of FloatingCpus is set sufficiently in advance

#### **16. Visualisation and human intervention.**

So finally we get to **D. The need to be able to visualise what is happening and verify that it is working as designed.**

Visualising job selection and backfill activity is done with a single display called ll\_jobs. Please note - ll\_jobs is a display program and does no data gathering. A perl program called nll\_sched provides the information displayed. nll\_sched runs every 10 seconds and uses the Loadleveler API to get the data needed then processes the data.

ll\_jobs has 2 modes - non-operational and operational, where operational is triggered by FloatingCpus <936. By watching the ll\_jobs display over time, it is possible to be confident that what is supposed to happen actually happens. Importantly when resources have been reserved, but the display shows CPUs are not going to be released because the active jobs will run for too long, detailed information is displayed from which action can be taken. And when action needs to be taken, it is shift staff taking the action. They make sure the nodes get released before a delay of more than 15 minutes occurs.

So next a display of ll\_jobs in non-operational mode and other related information.

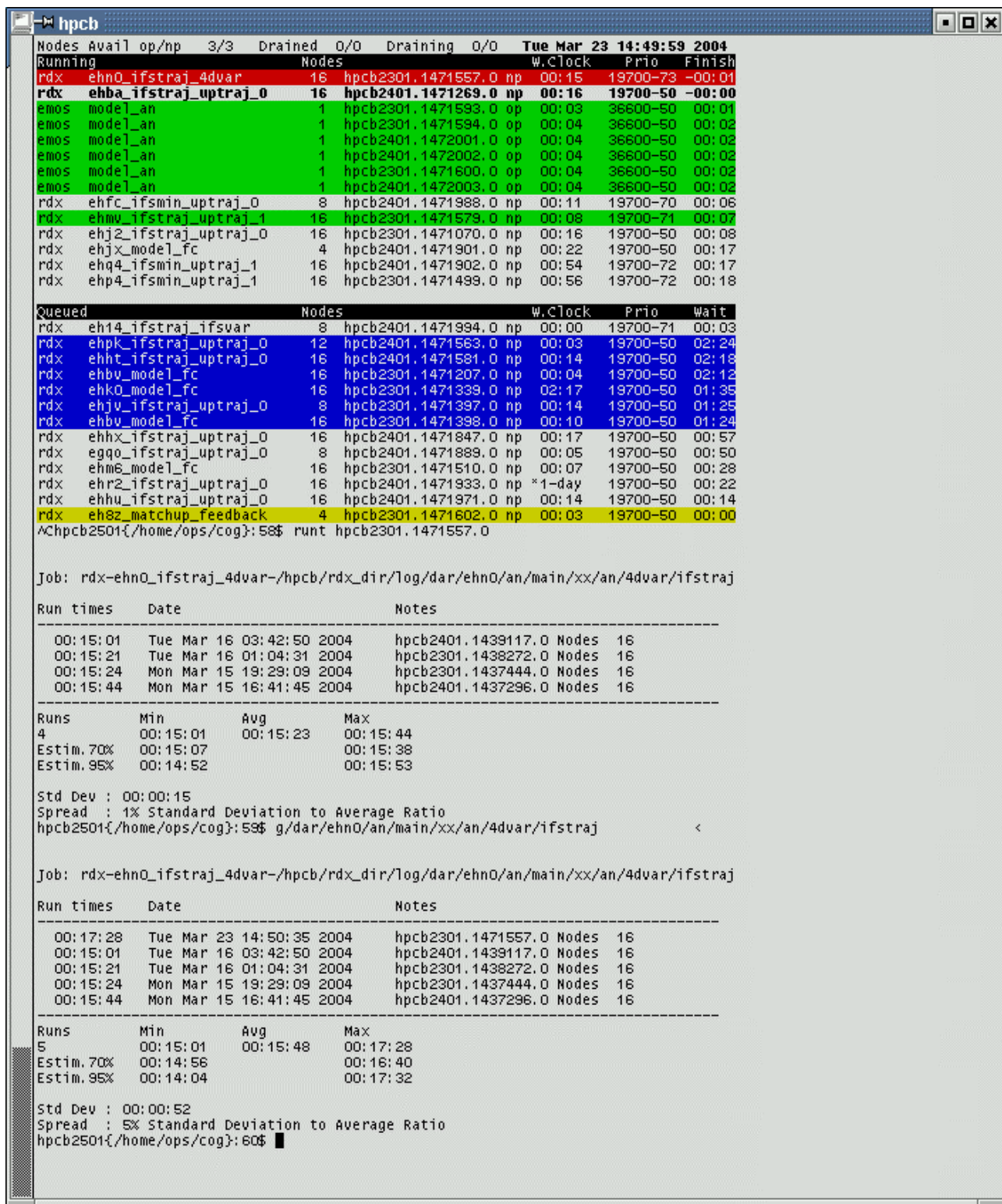


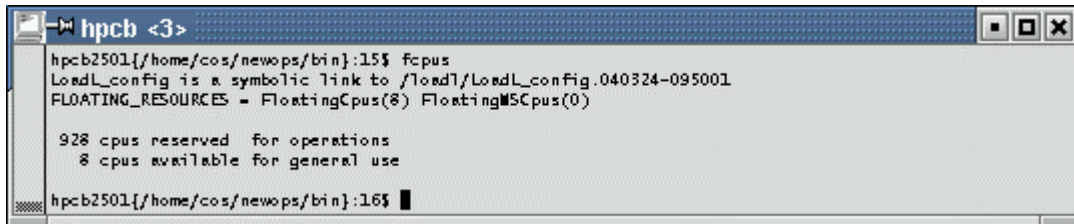
Figure 7 - ll\_jobs in non-operational mode (with no mention of nodes reserved), where Running jobs, Queued jobs and 2 sets of predicted run-times for job ehno\_ifstraj\_4dvar are shown.

Figure 7 (the top half) shows ll\_jobs in a non-operational period – Running jobs first, below this Queued jobs. No mention is made of nodes being reserved. Running jobs are listed top down in the order that they are expected to complete. Some jobs are highlighted in colour. The job in red has run for longer than predicted. Jobs in green started in the previous 3 minutes. Queued jobs are listed in the order they are expected to be dispatched according to SYSPRIO and UserPriority. Jobs in blue have been waiting for more than 1 hour, jobs in yellow are new and were submitted in the previous 2 minutes.

As there is a job that has run for longer than predicted I have extracted the original 4 run-times from which a predicted run-time of 00:15:53 was calculated and the 5 run-times, from which a new predicted run-time of 00:17:32 is calculated.

## 17. How many FloatingCpus have been reserved?

As mentioned before during operational periods, the value of FloatingCpus will be less than 936. The scripts fcpus can be used to check the value of FloatingCpus and the relationship with CPUs reserved.



```
hpcb2501[/home/cos/newops/bin]:15$ fcpus
LoadL_config is a symbolic link to /load1/LoadL_config.040324-095001
FLOATING_RESOURCES = FloatingCpus(8) FloatingM5Cpus(0)

928 cpus reserved for operations
8 cpus available for general use

hpcb2501[/home/cos/newops/bin]:16$
```

**Figure 8 - a display showing the relationship between the value of FloatingCpus set a CPUs reserved.**

When FloatingCpus is set to 8, 928 CPUS (116 nodes) are reserved (ask me later why FloatingCpus is not set to 0), and as the value of FloatingCpus is 8 (<936) ll\_jobs will show more information -see Figure 9 – below.

```

hpcb
Nodes Avail op/np 2/2 Drained O/O Draining O/O Wed Mar 24 10:13:34 2004
Running
emos prodgen_042 1 hpcb2401.1475471.0 op 00:02 36600-90 00:00
emos prodgen_045 1 hpcb2401.1475474.0 op 00:02 36600-90 00:02
rdx ehht_ifstraj_uptraj_0 16 hpcb2401.1475089.0 np 00:14 19700-80 00:04
rdx ehba_ifstraj_uptraj_0 16 hpcb2301.1474748.0 np 00:16 19700-80 00:07
emos 02_pf 4 hpcb2301.1475052.0 op 00:26 36600-70 00:21
emos 01_pf 4 hpcb2401.1475454.0 op 00:26 36600-70 00:21
emos 08_pf 4 hpcb2301.1475055.0 op 00:26 36600-70 00:21
emos 03_pf 4 hpcb2401.1475455.0 op 00:26 36600-70 00:21
emos 04_pf 4 hpcb2301.1475053.0 op 00:26 36600-70 00:23
emos 06_pf 4 hpcb2301.1475054.0 op 00:26 36600-70 00:24
emos 05_pf 4 hpcb2401.1475456.0 op 00:26 36600-70 00:24
emos model_fc 36 hpcb2401.1475421.0 op 01:12 36600-70 00:44
rdx ehf9_model_fc 16 hpcb2401.1474974.0 np *1-day 19700-50 22:52

Queued
emos control_cf 6 hpcb2401.1475462.0 op 00:41 36600-71 00:04
emos 07_pf 4 hpcb2401.1475457.0 op 00:26 36600-70 00:04
emos 09_pf 4 hpcb2401.1475458.0 op 00:26 36600-70 00:04
emos 11_pf 4 hpcb2401.1475460.0 op 00:26 36600-70 00:04
emos 12_pf 4 hpcb2301.1475058.0 op 00:26 36600-70 00:04
emos 10_pf 4 hpcb2301.1475059.0 op 00:26 36600-70 00:04
emos 13_pf 4 hpcb2301.1475067.0 op 00:26 36600-70 00:02
emos 15_pf 4 hpcb2401.1475470.0 op 00:26 36600-70 00:02
emos 14_pf 4 hpcb2301.1475068.0 op 00:26 36600-70 00:02
emos 17_pf 4 hpcb2401.1475461.0 op 00:26 36600-69 00:04
rdx ehq4_ifsmin_uptraj_0 16 hpcb2301.1475017.0 np 00:21 19700-70 00:27
rdx ehmv_ifstraj_uptraj_0 16 hpcb2301.1474687.0 np 00:20 19700-50 03:45
rdx eh8z_ifstraj_uptraj_0 16 hpcb2301.1474746.0 np 00:17 19700-50 03:30
rdx ehj8_model_fc 16 hpcb2301.1474789.0 np 01:55 19700-50 03:13
rdx ehpk_ifstraj_uptraj_0 12 hpcb2301.1474995.0 np 00:03 19700-50 01:15
rdx ehf9_model_fc 16 hpcb2401.1475419.0 np *1-day 19700-50 00:28
rdx ehhx_model_fc 16 hpcb2401.1475469.0 np 02:13 19700-50 00:02

Operations 116 nodes reserved at 10:05 for use at 10:09 Free Need
In op use 66 2 48
Chkpt ehf9_model_fc hpcb2401.1474974.0 16 32 10:06
You decide ehht_ifstraj_uptraj_0 hpcb2401.1475089.0 16 16 10:17
ehba_ifstraj_uptraj_0 hpcb2301.1474748.0 16 0 10:21
^Chpcb2501{/home/cos/newops/bin}: 93$

```

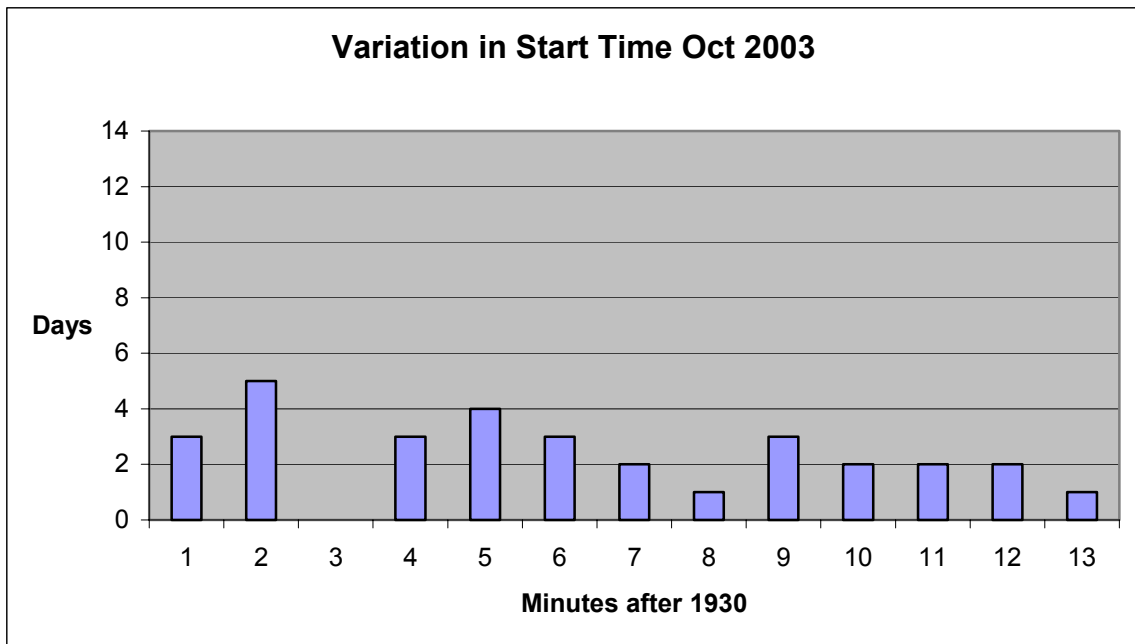
Figure 9 - `ll_jobs` during an operational period. `FloatingCpus = 8`, so additional information is provided particularly as resources will not be released at the time specified.

Please note the situation shown in Figure 9 was created artificially. Resource Reservation at 09:50 was delayed until 10:05. `ll_jobs` shows the time is 10:13:34, 116 nodes reserved, where 10:09 is the time the nodes were needed operationally. Unusually 48 nodes remain in use by non-operational jobs. One decision is easy. Job `hpcb2401.1474974.0 (ehf9_model_fc)` can be check-pointed and rerun. The other 2 jobs will complete soon, one at 10:17 (in 4 minutes time) and the other at 10:21 (in 8 minutes time), so waiting a few minutes for these 2 jobs to end will not cause a problem, both being well within the 15-minute flexibility.

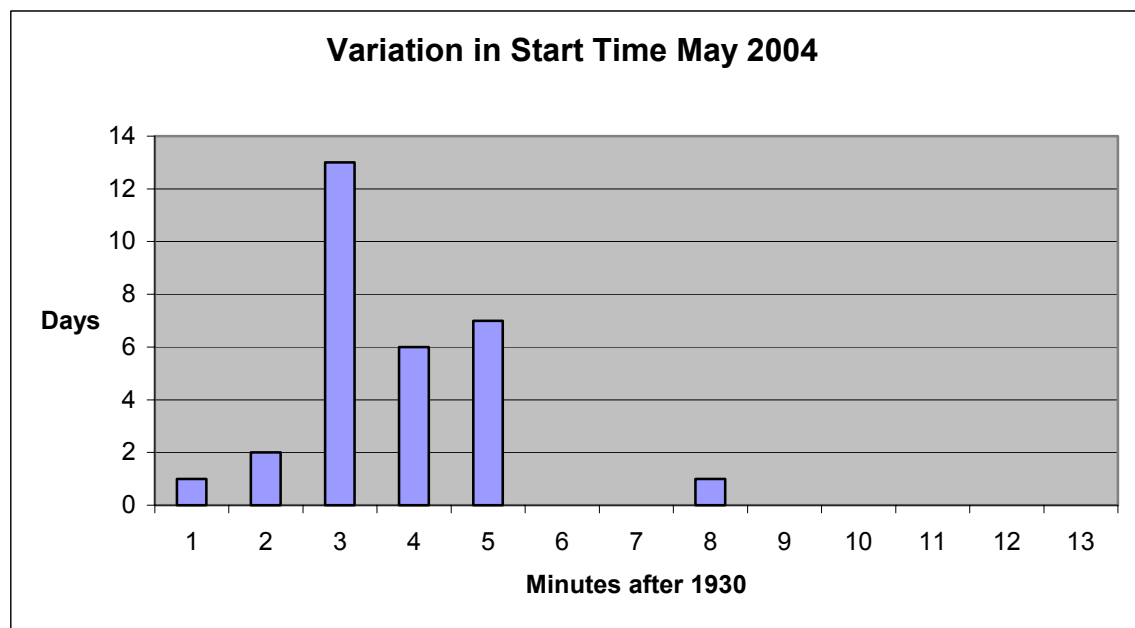
## 18. What has been achieved so far?

Has the variability in the start-times and end-times of the operational runs been reduced? Do the daily runs of the suite complete within 15 minutes of the optimum time?

The variability of the start-time of the operational runs, before FloatingCpus were reserved, is shown in Figure 10 and the variability of the start-time of the operational run after FloatingCpus were reserved is shown in Figure 11.



**Figure 10 - the variability in the start-time of operational runs, before Resource Reservation was introduced.**

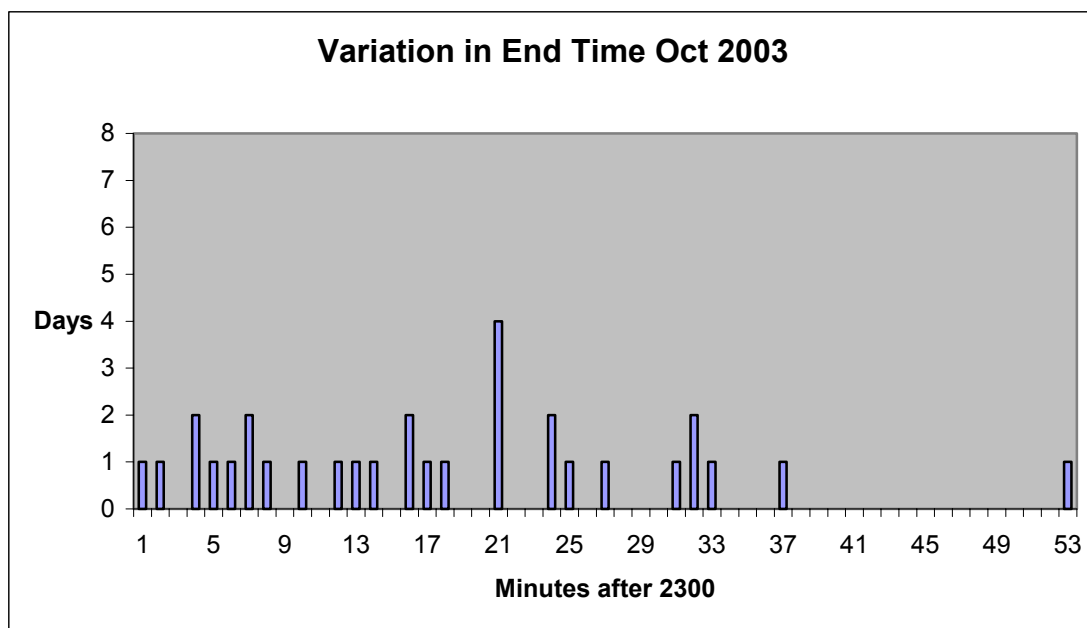


**Figure 11 - the variability in the start-time of operational runs before Resource Reservation was introduced.**

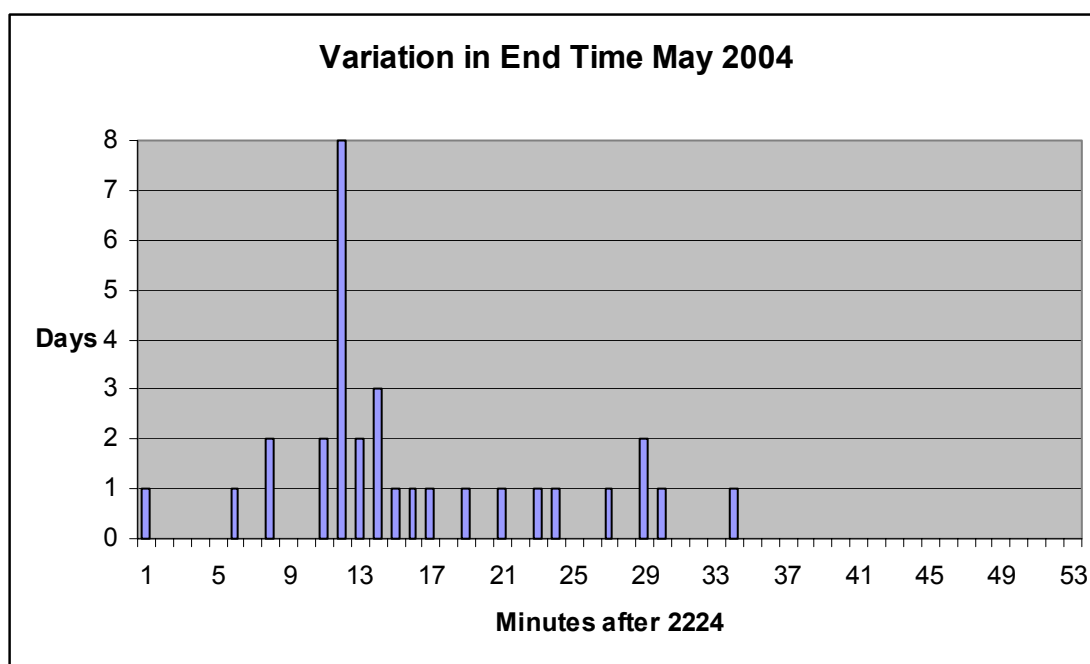
Comparing Figures 10 and 11 you can see the reservation scheme has significantly reduced the variability in the scheduling of the first operational job. Only once in May 2004 was more than 5 minutes of the flexibility used.

In Figures 12 and 13 you can see the variability of the end-times of the operational runs before FloatingCpus were reserved and after FloatingCpus were reserved.





**Figure 12 - the variability in the end-times of operational jobs, before Resource Reservation was introduced.**



**Figure 13 - the variability in the end-time of operational jobs, after Resource Reservation was introduced.**

Comparing figures 12 and 13 you can see the variability in the end-times of all operational jobs have been reduced too.

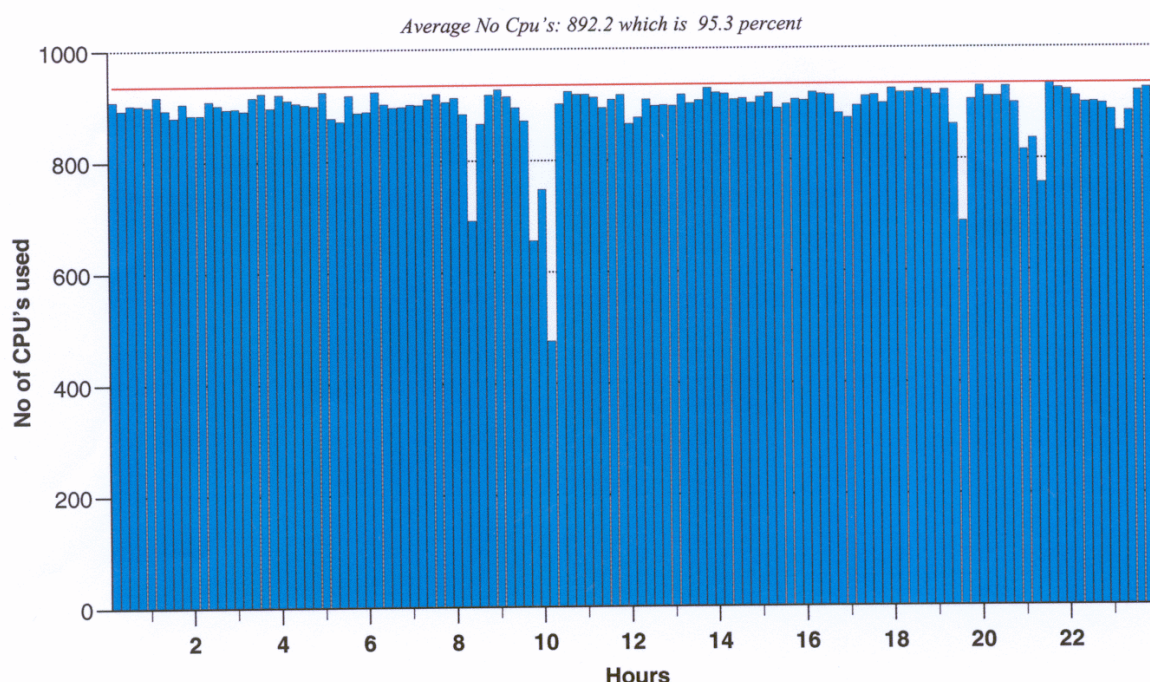
Note – since October 2003 significant improvements have been made to the efficiency of operational jobs and they run faster than before. As a result, the optimal end time of 23:00 achieved in October 2003 has been replaced by an end-time of 22:24 in May 2004. This means the need to keep operational jobs to a 15-minute variation is less acute at present, but not for long. Plans for higher quality forecasts (with greater computational needs) will soon push the end-time back to 23:00.

So clearly the variation exceeds 15 minutes and more work is needed to limit the variation to 15 minutes. 2 solutions are already planned, to optimise the pre-selection of jobs and to ensure when human intervention is needed that action is taken quickly.

Finally Figure 14 shows the allocation of CPUs to all jobs on a typical day since Resource Reservation was introduced.

## *CPU's allocated on HPCB by all parallel jobs*

*Thu 27 May 2004*



**Figure 14 - the allocation of CPU's to all jobs on a typical day since Resource Reservation was introduced, plotted over 12-minute intervals, where the red line (936 CPUs) is the maximum number that parallel jobs can use.**

Comparing Figure 14 with Figure 3 it is clear the allocation of CPU's to all jobs since Resource Reservation was introduced has reduced by a little over 1%. This is greater than was hoped for. But as mentioned earlier there are plans to optimise the selection of jobs before operational periods, which is when the greatest waste occurs. So importantly not only will optimising job selection minimise runtime variation it can be made to increase node utilisation too.

## **19 The conclusion.**

The reservation scheme has been very effective in keeping the start-time to a tighter schedule but less so with the end-time. In addition there is a reduction to the overall system utilisation of just over 1% as seen when comparing Figure 3 and Figure 14. But ways have been described to both minimise the variation and maximise the allocation of nodes and work will continue to achieve this.

## **20 Main points.**

It is essential that parallel jobs can be scheduled independent of other systems, that the jobs are designed well (suit the system) and system performance (GPFS, I/O nodes, Loadleveler, network, paging) does not vary. Thus (using historical data) predicted run-times will be accurate.

The Loadleveler feature FloatingCpus enables resources to be controlled globally, dynamically and logically, independent of job class where jobs that request FloatingCpus are members of one logical class and jobs that do not request FloatingCpus are members of another logical class. Managing the resources to suit these 2 logical classes could not be easier.

Setting values of FloatingCpus a fixed time in advance based on averages is very straightforward and reasonably effective but the variation in runtimes and better system utilisation could be obtained by optimising job selection prior to operational periods.

Reducing the value of FloatingCpus without turning classes off means that the underlying scheduling of non-operational jobs continues as normal in the reduced set of nodes, and there is complete flexibility as to which nodes are used by operational jobs and which nodes are used by non-operational jobs.

Having the system set `wall_clock_limit` with offset 1+ overcomes the kill on `wall_clock_limit` exceeded. Operational jobs have their `wall_clock_limit` set automatically too which means operators need not remember how long the multitude of operational jobs should take to run and any over-run is flagged immediately.

It is essential that monitoring tools (in our case operator displays) are created so that it is possible to confirm both backfill and scheduling are performing as expected or when events do not go to plan a mechanism (again operator displays) is in place to make operators, administrators and analysts aware of the problem(s).

There is no need to have lots of user batch classes (we had many NQS queues on previous systems) for jobs that are short, long, slow, fast, big, small etc so that in pre-operational periods the right jobs can be selected by draining classes, reducing job-limits per class, per user etc.

## **21 Other thoughts.**

The ability to schedule jobs as described stems from the understanding that the system used is a High Performance Cluster (HPC) System managed as a Super-Computer and is not a configuration of loosely coupled Server systems. All those involved in providing elements of the service on the HPC system; analysts, support staff, users, administrators, managers, have a fundamental desire for the system to be configured optimally and for jobs to run efficiently. I cannot stress enough that jobs must not use (need) swap space. The processes in each node of a multi-node, high performance, well parallelised, cpu-bound job, must not exceed physical memory as it only takes 1 process to use too much memory and all nodes will perform badly as the rogue process swaps.

Ideally jobs will parallelise well, scale reasonable well and use a whole node or multiples of nodes. Memory sharing is only permitted if job performance and known-runtimes are not altered. If users have CPU-bound jobs that use less than 1 node, particularly serial jobs with 1 process, the users are advised to run the jobs on Server systems.

Jobs that perform badly must be detected and users must know to call support staff for help (or eventually get caught) if they realise new jobs (for which there is no known run-time data) perform badly. Users and more importantly application programmers, who make it possible for suites of jobs (in the form of an experiment) to be submitted automatically, must be willing to help sort out job related problems and identify solutions.

As the release of all nodes for use by operational jobs occurs twice a day, all user jobs have to finish within 1 hour. Jobs that run for more than 1 hour should either create restart files so that they can be killed without substantial losses, or should have the ability to trap a signal and then create restart files before killing themselves. In Figure 9 such a job was labelled Chkpt. This job will accept and trap such a signal, write a restart file and kill itself.

There is no room for sloppy philosophy. The idea that it is OK for a job to be inefficient and run for a long time as long as the user pays for it, is just not acceptable.