Scheduling multicore workload on shared multipurpose clusters

J A Templon,1, C Acosta-Silva2,4, J Flix Molina2,3, A C Forti5, A Pérez-Calero Yzquierdo2,3, and R Starink1

1 Nikhef, Science Park 105, 1098 XG Amsterdam, NL
2 Port d’Informació Científica (PIC), Universitat Autònoma de Barcelona, Bellaterra (Barcelona), Spain
3 Also Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), Madrid, Spain
4 Also at Institut de Física d’Altes Energies, IFAE, Edifici Cn, Universitat Autònoma de Barcelona, Bellaterra (Barcelona), Spain
5 School of Physics and Astronomy, University of Manchester, Oxford Road, Manchester, UK, M13 9PL
E-mail: templon@nikhef.nl

Abstract. With the advent of workloads containing explicit requests for multiple cores in a single grid job, grid sites faced a new set of challenges in workload scheduling. The most common batch schedulers deployed at HEP computing sites do a poor job at multicore scheduling when using only the native capabilities of those schedulers. This paper describes how efficient multicore scheduling was achieved at the sites the authors represent, by implementing dynamically-sized multicore partitions via a minimalistic addition to the Torque/Maui batch system already in use at those sites.

The paper further includes example results from use of the system in production, as well as measurements on the dependence of performance (especially the ramp-up in throughput for multicore jobs) on node size and job size.

1. Introduction

Computer scientists have studied the problem of task scheduling in depth. For cases in which the task arrival times and resource requirements (e.g., task duration) are known in advance, scheduling is a solved problem [1]. For tasks common to grid infrastructures like the Worldwide LHC Computing Grid [2] (WLCG), the European Grid Infrastructure (EGI) [3], or a national grid infrastructure, neither are known in advance. The problem considered here, optimal scheduling on a cluster connected to one or more general-purpose grid infrastructures, belongs to the most general scheduling case, and is an unsolved research problem [4]. The work we present is motivated by ideas from statistics, and is inspired by earlier work on grid scheduling done at Nikhef [5–7]. Measurements presented here were made at Nikhef unless otherwise stated.

2. Problem Description

Tasks ("jobs") arrive to clusters (computing resources at a single geographical grid "site") from several (potentially hundreds) of different locations and users, hence the inability to predict the
arrival time distribution. Resource consumption estimates are generally not submitted with the jobs; by publishing the maximum time allowed, sites make an implicit guarantee that the job will be allowed to take up to this time.

There is almost always work waiting to be done; the occupancy of the system is 97.8% over the last four years, and typically jobs are being run from about seven different groups of users. Which seven groups, and their workload volumes, change all the time. Static allocation of nodes to specific groups leads to frequent cases of nodes doing nothing.

As long as the jobs’ resource requirements are ”standard” there is no problem; all jobs fit equally well into all slots. When a job slot is freed by an exiting job, the scheduler picks a job to replace it. How long the job will run is not important, as long as it finishes within the slot reservation (typically a couple of days).

The advent of multicore jobs changed the game. Our multicore jobs require cores co-located on the same physical node, as the point of being multicore is to reduce job memory consumption by sharing static memory across processes; this only works if all processes run on the same physical machine. In order to run such a multicore job, one must wait for several slots to free up, all on the same physical node. This process is called ”draining”.

Suppose a request for a multicore job arrives, asking to run a single job with N cores on a single node. In the optimal scenario, draining goes like this:

identify the node, which, based on available information like remaining running times of current jobs, is predicted to be the first to release N job slots

capture these job slots as they are released

backfill these slots; compute (again based on available information) how long it will be until the Nth slot will become free, find waiting jobs needing less time than this, and run those jobs in the captured slots. This prevents wastage of the captured slots while waiting for the last, Nth slot, to free up.

run the multicore job when the Nth slot frees up, by which time all the backfilled jobs will have also completed.

In our grid systems this approach fails:

• prediction of the fastest-to-drain node fails, as the available information is useless (on average wrong by factors of 3 to 10)

• very few of the jobs submitted declare a short run time, so even if we accept a suboptimal choice of the node to be drained, we cannot backfill the captured slots.

The approach and system we describe in this paper is intended as a solution to the above problem. Taking the jobs we receive as an unchangeable given, we are faced with accepting the realities of node draining without backfilling. The problem is then turned around, and addressed by our approach: the challenge is to minimize the amount of draining, both reducing the frequency of slot draining as well as reducing the time needed to drain.

3. Limiting the amount of draining: the floater
Our technique for limiting the amount of draining is to only do it once. Multicore slots are created by draining; draining is initiated by moving nodes from a ”normal (single-core) pool” to a ”multicore” pool; nodes in the multicore pool are only allowed to run multicore jobs, so the draining only needs to be done once. This “once” does presuppose that all multicore jobs are of the same size (same number of cores). Our original implementation was based on eight-core slots, this was later changed to four cores per slot.

The flow of multicore jobs is not steady, and sometimes single core jobs from other groups have higher priority, so the boundary between the multicore and single core pools is not static;
it can be adjusted simply by moving nodes from one pool to the other. This is the job of the “floater”, the name coming from the little floating device in any flush toilet that controls the water valve based on the water level within the tank. The floater is implemented as a small program which runs in a cron job, inspects the ”levels” of running multicore jobs, waiting multicore jobs, and number of empty slots in “draining” status. Based on those levels it decides whether to open the ”valve” and add (or remove) one or more nodes from the multicore pool.

3.1. Algorithm

The algorithm for the implementation, mcfloat$^1$, is almost as simple as that of the floater in a toilet tank. Our experience is that sophisticated scheduling algorithms are in most cases only slightly better than simple ones, despite requiring much more coding effort. For all algorithms there are cases for which they perform poorly and/or exhibit undesirable behavior; sophisticated algorithms tend to have more of such cases than simple algorithms. Hence we only improved the algorithm described here in those cases where the production performance data show that more sophistication is warranted. We omit here details that are specific to the particular batch system and scheduler used (Torque and Maui, from Adaptive Computing). For those details, please refer to the source code.

As input, mcfloat requires a specification of which worker nodes are eligible to be included in the multicore pool. It also requires the batch system and scheduler to be set up such that multicore jobs submitted to the site, are eligible to run on nodes in that multicore pool, and more importantly, single core jobs are not allowed to run on nodes in the multicore pool. On each iteration, mcfloat checks the number of multicore jobs waiting to be run, the total number of slots in draining status (i.e. empty but not yet eligible to run a multicore job), and for each node, a check is made for the existence of ”empty” multicore slots – these are slots which are already drained and could immediately run a multicore job, but (for whatever reason) the scheduler has left empty. The information from this check is persistent between runs, i.e., mcfloat knows if a ”empty” multicore slot was present on the same node for two runs in a row.

After collecting this information, mcfloat takes the following steps:

(i) if persistently empty multicore slots were found, move half of the nodes with such slots (at least one node) from the multicore pool back into the main pool. Then exit.
(ii) otherwise, check the number of unused (draining slots). If the number is below the acceptable level:
   (a) check if there are any multicore jobs waiting to be run
   (b) if so, check if the number of draining nodes is acceptable; if so
   (c) grab one or more candidate nodes (at random) out of the single-core pool and add to the multicore pool (unless all candidate nodes are already in the multicore pool)
(iii) if the number of draining slots exceeds the acceptable level
   (a) move nodes back into the single-core pool until the number of empty slots inside the multicore pool is back at acceptable levels.

The most complex part of the algorithm is the choice, in (i) and (iii), of which nodes to remove from the multicore pool. The original algorithm was too simple; for example, we observed mcfloat removing a single, almost-fully-drained node with e.g. 7 empty slots, instead of removing 7 recently-added nodes, each with a single empty core and no multicore slots created yet. Clearly it would have been better to remove the 7 slowly-progressing nodes.

We formulate the removal problem in this way:

$^1$ The code is available from the Nikhef Physics Data Processing Group’s SVN repository; the version described in this article can be accessed via http://ndpfsvn.nikhef.nl/cgi-bin/viewvc.cgi/pdpsoft/nl.nikhef.ndpf.mcfloat/trunk/mcfloat?revision=2786
how do we remove nodes such that at least M unused slots are removed from the multicore pool, while minimising the number of running multicore jobs that are removed from the pool?

This is closely related to the "knapsack problem" [8] that in most cases has no analytic solution. Our approximate solution is as follows:

(i) `mcfloat` first removes nodes in the multicore pool for which no multicore slots have been created. Nodes with the fewest empty slots are removed first. The rationale here is that it is better to move nodes that are making little draining progress first. If we are attempting to run 8-core jobs, a node with 7 empty slots is "almost ready" and it is better to preserve this one by instead removing 7 nodes each containing only one empty slot.

(ii) If removal of all the above nodes is not sufficient, nodes on which multicore jobs are running are ranked by the ratio (unused slots)/(running multicore jobs). Nodes are removed from the multicore pool by rank, starting with the highest rank.

This algorithm is close to optimum. What it misses is the occasional case in which it would have been better to pick a node with a lower (unused slots)/(running multicore jobs) ratio, as the node with the highest ratio has more unused slots than needed to be removed.

4. Performance

`mcfloat` has been in operation at Nikhef since April 2014, PIC and Manchester shortly thereafter. The system performs adequately as designed, as shown in figures 1 and 2.

There are only a few parameters to tune in the system:

- how large to allow the pool to grow (list of candidate nodes)
- how often to check whether nodes should be added or removed
- how many unused slots to tolerate

The maximum pool size is highly site-dependent; typical choices for the other two parameters are to check every 10 minutes, and to allow 1.5% of the farm’s slots to be draining (unused).

![Figure 1. mcfloat performance at Nikhef for a period of 12 days. Orange represents the number of cores running multicore jobs; green shows single-core jobs still running on draining pool nodes; grey (barely visible) are unused cores on draining nodes. The pool was fully drained (completely orange) for several days here. For a few hours on Saturday 04 there were no multicore jobs being submitted to Nikhef, hence the decrease of the pool size to close to zero.](image)

5. Considerations on Efficient Draining

Figures 1 and 2 show a slow draining process. In figure 1 (Nikhef), it took 2.5 days to drain 1000 cores; typical performance is 5.5 days to go from an empty multicore pool, to a fully-drained multicore pool running at maximum capacity. In this section we present some considerations on
how to maximise the draining rate, without increasing the fraction of unused (draining) slots. One other consideration is worth stating: since draining is expensive, loss of pool nodes due to external reasons should be avoided where possible. For example, the CREAM CE that fronted the multicore pool occasionally stopped accepting submissions. This is a standard CREAM feature that avoids overloading a CE during periods of heavy job submission. Such overload protection can prevent multicore submission for up to an hour, sufficient to cause loss of the majority of the pool nodes (see figure 1). The solution in this case was to allow a second CE to also submit jobs to the multicore queue; the chance that both CEs are "down" simultaneously is quite small in practice.

In the beginning of this project, a WLCG-wide decision was made to start with 8-core jobs. At Nikhef, these jobs could be run on three worker-node classes: 8, 12, and 32 cores. Experiments were run with 4-core jobs as 8-core jobs do not fit well with 12-core nodes. We found that draining speed increased with increasing node size (more cores per node) as well as with decreasing job size (fewer cores per job).

The observations can be understood in detail by observing the populations of jobs on production worker nodes. We used the following method to do so: at a particular time $t$, a snapshot was taken of the batch system: the state of all nodes, whether they were in the multicore or general pools, and for each node, the jobIDs of all jobs running on that node. Waiting a few days, all of those jobs have completed, and the batch system logs tell us how long each job on each node kept running after time $t$; this gives us a draining profile for each worker node in the system. An example is shown in figure 3.

Given such draining profiles, we can explore how the growth of the multicore pool capacity
depends on node size and job size. We show the two extreme cases in figures 4 and 5; we also provide a table with the entire matrix of machine and job sizes.

Figure 4. Draining profiles for 8-core nodes and the case of 8-core jobs. The leftmost profile is that for the worker node with the shortest time to complete draining (7.9 hours), at which point it can run an 8-core job. Rightmost shows the node which takes the longest to drain completely (33 hours). The middle plot shows the result of averaging the drain profile over all the main-pool 8-core nodes (20.7 hour average drain time).

Figure 5 shows the worst case (slowest pool growth): trying to run 8-core jobs on 8-core nodes. The fastest node is drained in 7.9 hours, the slowest in 33 hours. Figure 5 shows the best case (quickest pool growth) that we explored: running 4-core jobs on 32-core nodes. The quickest node can already run a 4-core job after 1.3 hours of draining, and the worst case is 3.3 hours. The complete matrix is shown in table 1.

Figure 5. Draining profiles for 32-core nodes and the case of 4-core jobs. The leftmost profile is that for the worker node with the shortest 4-core drain time (1.3 hours). Rightmost shows the node with the longest 4-core drain time (3.3 hours). The average node (middle) takes 2.4 hours to drain a 4-core slot.

The differences shown in table 1 can be explained by combinatorics. The remaining-time distribution of all main-pool jobs at time $t$ shows that 12% of them take longer than 20 hours.
Table 1. The average time to create a multicore slot of either 8 or 4 cores, by draining nodes with various core counts. The dataset is the same that generated the draining profiles above. Multicore slots are created more quickly for smaller jobs and larger nodes.

| Job Size | average time to drain N cores (hr) |
|----------|-----------------------------------|
|          | 8-core node | 12-core node | 32-core node |
| 8 cores  | 21          | 10           | 4            |
| 4 cores  | 7           | 5            | 2            |

Assuming jobs land on nodes at random, almost all nodes with 8 cores will host at least one job that will run for another 20 hours. A node with 32 cores will host about four such jobs on average (and figure 5 verifies this argument). The difference is that for an 8-core node and 8-core jobs, all 8 slots need to drain before we can start a job, so we are stuck with the 20+ hours. For the 32-core node job, the first 8-core job can already start long before those four 20+ hour jobs have ended.

One final consideration is that smaller jobs allow not only for faster acquisition of drained slots, but also for more aggressive draining in general. This is illustrated in figures 6 and 7, which are based on the data of figure 3.

The figures show the same draining profile; the orange area corresponds to multicore work that starts when the node has been sufficiently drained. On the left side, the multicore jobs require 8 cores. The white area corresponds to the unused, draining slots. The figure on the right is the same, except the multicore jobs require 4 cores. The first job can already start at about 9 hours, and the amount of white area (unused cores) shrinks by about a factor of 3. For four-core job operation, jobs start to run faster, and the amount of nodes in draining state can be tripled compared to the 8-core case. The particular improvement factor is dependent upon the population of jobs in the system; aside from exceptional circumstances (like a complete flush of all jobs in the system), the improvement for smaller jobs is large. This is illustrated in production in figure 8.
Figure 8. Activity profile for the multicore pool at Nikhef. On Thursday 26th, the switch was made from 8-core jobs to 4-core jobs. The change in multicore slot growth (orange area) is striking. The `mcfloat` algorithm was not modified.

6. Conclusions
The approach works well and the implementation performs adequately. Possible future work includes preferring candidate nodes with more cores, although this is partially taken care of by the sort ordering (larger nodes tend to have lower ratios). Another area is tighter scheduler integration, to enable return of cores to the main pool when needed to serve higher-priority jobs.

Acknowledgments
One of us (JT) would like to acknowledge helpful discussions with D. van Dok (Nikhef) on the knapsack problem and its solutions.

The Port d’Informaci Cientfica (PIC) is maintained through a collaboration between the Generalitat de Catalunya, CIEMAT, IFAE and the Universitat Autonoma de Barcelona. This work was supported partly by grants FPA2010-21816-C02-00 from the Ministry of Science and Innovation and FPA2013-48082-C2-1/2-R from the Ministry of Economy and Competitiveness, Spain. Additional support was provided by the EU 7th Framework Programme INFRA-2010-1.2.1: Distributed computing infrastructure Contract Number RI-261323 (EGI-InSPIRE).

References
[1] See the description of the m/d/k queuing model (and references) on wikipedia: url: http://en.wikipedia.org/wiki/Queueing_theory
[2] Bird I 2011 Ann. Rev. Nucl. Part. Sci. 2011. 61 99–118
[3] Newhouse S 2011 European grid infrastructure - an integrated sustainable pan-european infrastructure for researchers in europe (egi-inspire). permalink: https://documents.egi.eu/document/201
[4] See for example the wikipedia article for a slightly less general case than described here: url: http://en.wikipedia.org/wiki/G/G/1_queue
[5] Li H, Groep D L, Templon J and Wolters L 2004 IEEE/ACMIntl. Symp. on Cluster Computing and the Grid (CCGrid) (IEEE Computer Society) pp 301–308
[6] Li H, Groep D L and Wolters L 2004 Proc. 6th IEEE/ACM Int. Workshop on Grid Computing (IEEE Computer Society) pp 234–241
[7] Templon J 2008 Ert, glue, and scheduling: Now and later. talk at Enabling Grids for E-scienceE 2008. url: http://indico.cern.ch/event/32220/session/78/contribution/194
[8] In the knapsack case, one would be looking for a the maximum number of unused slots that could be dumped, given a certain limit on the number of running multicore jobs that may be removed. url: https://en.wikipedia.org/wiki/Knapsack_problem