IMPROVING THE LOAD OF SUPERCOMPUTERS BASED ON JOB MIGRATION USING CONTAINER VIRTUALIZATION

S.P. Polyakov [a], Yu.Yu. Dubenskaya

Skobeltsyn Institute of Nuclear Physics, M.V.Lomonosov Moscow State University (SINP MSU), 1(2), Leninskie gory, GSP-1, Moscow 119991, Russia

E-mail: [a] s.p.polyakov@gmail.com

Modern supercomputer schedulers on average may leave ~10% and sometimes as much as 30% of the computational resources idle. One possible approach to increase the load is to use an additional queue of low-priority jobs small enough to fit into the schedule gaps. We propose to use this approach for non-parallel jobs with arbitrary runtime wrapped in containers to allow them to be saved and migrated to other nodes or back to the queue. As a result, all the idle nodes can be used for computations. We also estimate the increase in average load and utilization efficiency that can be achieved using this approach.

Keywords: supercomputers, supercomputer schedulers, average load, containers, container virtualization, container migration

1. Improving the load of supercomputers and containers

Modern supercomputers use schedulers to allocate the computational resources. Despite the substantial effort put into developing and improving scheduler algorithms, the average load of supercomputers is often close to 90% and can be as low as 70% [1, 2]. This is caused by varying dimensions of the submitted jobs (number of required computational nodes, runtime), unpredictable submission time, and inaccuracy of runtime estimates.

Besides further improving the scheduling algorithms, other approaches can be used to address the problem. One example is allowing the idle nodes to be used by additional low priority jobs that can be terminated whenever the scheduler assigns the node to a regular job. The example of this approach is the opportunistic use of idle Titan-2 resources by ATLAS [3].

Numerous problems can be solved by computations without parallelization so the corresponding jobs can use any available resource slot. However, many such jobs require substantial computation time and cannot fit into smaller gaps in the schedule. This problem can be solved by an efficient mechanism of saving the current state of a computational node and continuing computations from the saved state, possibly on a different node.

A variation of such mechanism, a live container migration, is implemented in several container virtualization platforms including OpenVZ [4] and Docker [5] (in Docker, live migration is currently available in experimental mode only).

We propose to increase the load of supercomputers by using an additional queue of non-parallel jobs wrapped in containers. Containers can be started very quickly and impose little to no overhead. Using the live migration tools containers can be saved and returned to the queue or migrated directly to other nodes before the allotted time is over. Assuming the minimal scheduler time slot is sufficient to start a container, perform some computations, and save it, the proposed approach potentially allows to use all the nodes left idle by the scheduler and increase the load to 100% of the available computational nodes. We are currently developing a prototype of the job management system implementing this approach.

Our proposed system will not eliminate the need for further improvements in scheduling algorithms: first, the containerized jobs that often need to be stopped and restarted reduce the computational nodes efficiency, and second, these jobs are presumed to have lower priority than the regular jobs. Conversely, improvements of the scheduling algorithms will reduce the effect of the proposed system but will not make it useless while the load remains substantially lower than 100%.

In the following section, we further discuss scheduling algorithms and find an estimate of the potential load increase resulting from our approach.

2. Scheduling algorithms and load estimate

One of the commonly used basic scheduling algorithms is FCFS (First-Come First-Served) with backfilling [6]. The algorithm schedules the jobs in order of submission until the first job that cannot be started immediately. This job is then assigned a reservation at the earliest time slot when enough computational nodes become available, and the following jobs are only allowed to use these nodes if they are expected to finish before the reserved time. A number of variations and modifications have been proposed since the algorithm first appeared, and modern schedulers such as SLURM [7] allow to tune the algorithm using various optimization parameters.

Figure 1 illustrates the problem and the backfilling approach: jobs 4-7 cannot be started immediately so they are scheduled to further slots, in some cases changing the execution order. As illustrated by the job 7, the nodes executing the same job do not have to be adjacent, although some schedulers allow to take locality into account. Note that the backfilling algorithm used in Figure 1 allocates multiple reservations. A variation of the algorithm that allocates reservations to all jobs is called conservative backfilling [8]; some schedulers such as MAUI [9] allow system administrators to set up the number of reservations.

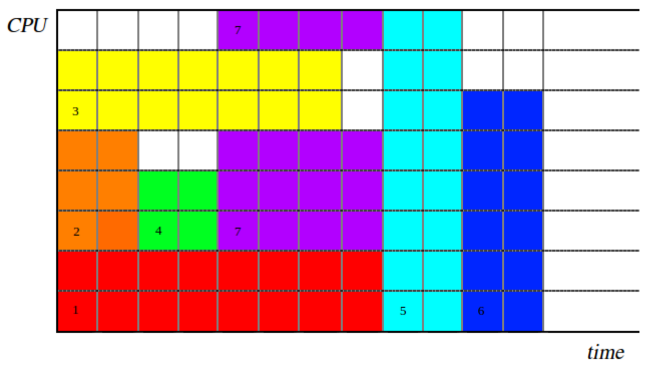

Figure 1. An example of a schedule

All variations of backfilling algorithm require an estimate of job runtime. Some schedulers require the estimate to be submitted by the user along with the number of requested computational units, the other may attempt to predict the runtime based on the previous jobs by the user [10, 11]. Inaccurate estimates may result in additional idle time for the nodes that can be used to run other jobs, or in the job failing to complete before the end of the allotted time. In the latter case, the scheduler may either terminate it forcibly or let it keep running and reschedule the affected reservations.

For our load estimate we used a simulation of a simple conservative backfilling algorithm. Job dimensions were generated randomly based on approximated 2017 data from Lomonosov-1 supercomputer (average number of CPU slots 7.5, standard deviation 16.7, maximum 512; average runtime 255 minutes, standard deviation 1204 minutes, maximum 15 days). For user runtime estimates, we used a simplified model with a single parameter, prediction accuracy *p*. The model is based on the observation [12] that users typically select one of the "round" values as their runtime estimates. We used a set of 20 standard round values (1, 2, 5, 10, 20, 30 minutes, 1, 2, 3, 6, 8, 12 hours, 1, 1.5, 2, 3, 5, 7, 10, 15 days) as possible inaccurate estimates. For each job one of the three runtime estimates was selected randomly:

(a) equal to actual runtime (probability p);

(b) runtime rounded up to the closest round value (probability $p(1-p)$);

(c) runtime rounded up to the next closest round value (probability $(1-p)^2$).

The simulations were run for a supercomputer with 512 CPU slots and 180 days (with a time slot corresponding to one minute). The calculated values of average load l were averaged over 10 attempts. Figure 2 shows the simulation results.

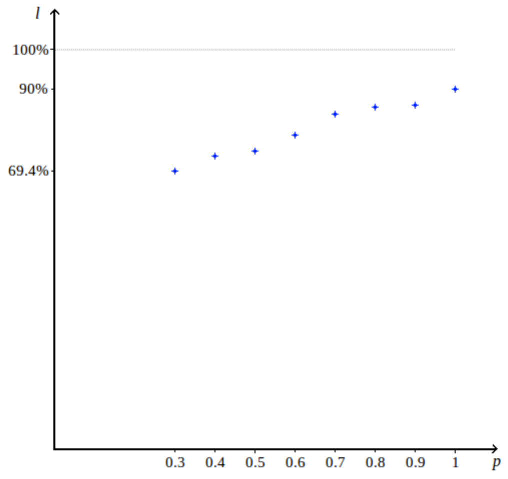

Figure 2. Average load estimates

If $T=1$ is the duration of a time slot used by the scheduler, $t<1$ is the combined time of starting and saving a containerized job, and c is the performance ratio between containerized and non-containerized jobs, then the increase of average load from 90% to 100% due to the use of our proposed system corresponds to the increase of average utilization efficiency at least by $0.1(1-t)c$ of the peak efficiency.

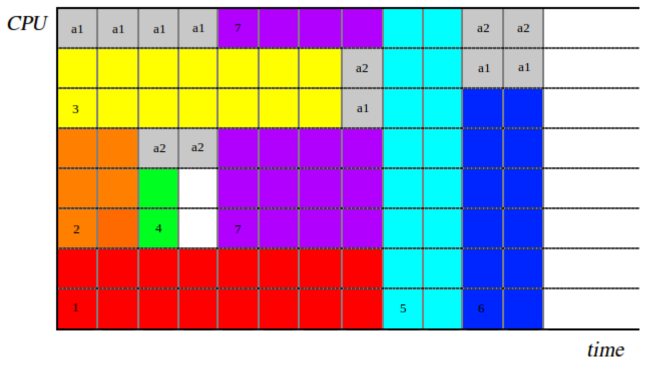

Figure 3. A schedule with added containerized jobs

This estimate assumes that all additional containerized jobs are only allowed to run for one time slot and must be saved before it is over regardless of the existing reservations. Alternative approach is allowing these jobs to run continuously and only saving and migrating them before the reservation by a regular job comes up. The potential downside of this approach is illustrated by the Figure 3: if job 4 terminates earlier than planned, job 7 can start earlier (or another job using 4 CPU slots and 1 time slot can be fit into the gap, if the job rescheduling is not allowed). But if the additional containerized jobs a1 and a2 are not saved by the end of the third time slot, they cannot be terminated without losing some progress. Similarly, a newly submitted job may be delayed if the nodes running containerized jobs do not have enough time to save the containers.

3. Acknowledgement

The work was supported by the Russian Foundation for Basic Research grant #18-37-00502 “Development and research of methods for increasing the performance of supercomputers based on job migration using container virtualization”. The authors would also like to thank Sergey Zhumatiy for useful discussions.

4. Conclusion

We proposed an approach to increase the average load of supercomputers using the container virtualization mechanisms. We also estimated the potential increase in average load resulting from this approach, and the corresponding increase in utilization efficiency.

References

[1] Leonenkov S., Zhumatiy S. Supercomputer Efficiency: Complex Approach Inspired by Lomonosov-2 History Evaluation // Russian Supercomputing Days: Proc. Int. Conf. (Sept. 24-25, 2018, Moscow, Russia). Moscow State University, 2018. pp. 518–528.

[2] Antonov A.S. et al. Examination of supercomputer system jobs flow dynamic characteristics // Computational methods and programming. 2013. V. 14, no. 4. pp. 104–108 (in Russian).

[3] F. Barreiro Megino et al. [ATLAS Collaboration]. Integration of Titan supercomputer at OLCF with ATLAS Production System // J. Phys.: Conf. Ser. 2017. V. 898, no. 9. P. 092002.

[4] http://openvz.org/

[5] https://docker.com/

[6] Lifka D. The ANL/IBM SP scheduling system // 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). Feitelson D.G., Rudolph L. (eds.). Springer-Verlag, Apr. 1995. Lect. Notes Comput. Sci. V. 949. pp. 295–303.

[7] https://slurm.schedmd.com/

[8] Mu'alem A.W., Feitelson D.G. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling // IEEE Trans. Parallel & Distributed Syst. June 2001. V. 12 issue 6. pp. 529–543.

[9] http://www.adaptivecomputing.com/products/open-source/maui/

[10] Tsafrir D, Etsion Y., Feitelson D.G. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates // IEEE Trans. Parallel & Distributed Syst. June 2007. V. 18 issue 6. pp. 789–803.

[11] Gaussier E. et al. Improving backfilling by using machine learning to predict running times // Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis. ACM, 2015. P. 64.

[12] Tsafrir D., Etsion Y., Feitelson D.G. Modeling user runtime estimates // 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). Feitelson D.G. et al. (eds.). Springer-Verlag, June 2005. Lect. Notes Comput. Sci. V. 3834. pp. 1–35.