A Container-Based Job Management System for Utilization of Idle Supercomputer Resources*

Stanislav Polyakov*[0000−0002−8429−8478], Julia Dubenskaya[0000−0002−2437−4600], and Elena Fedotova[0000−0001−6256−5873]

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

Abstract. We propose a system for utilization of idle computational resources of supercomputers. The system executes additional low-priority jobs inside containers on idle nodes and uses container migration tools to interrupt the execution and resume it later, possibly on different nodes. We implemented a prototype of the proposed system for Docker containers and demonstrated that it performs the necessary operations successfully. Our experiments based on simulation show that the proposed system can increase the effective utilization of supercomputer resources, in some cases by as much as 10%.

Keywords: Data processing · Supercomputer scheduling · Average load · Container · Container migration.

1 Introduction

Supercomputers are very expensive to build and maintain, so it is important to prevent any unnecessary losses in their performance. In many cases average load of supercomputers is significantly lower than 100%. For example, Lomonosov and Lomonosov-2 supercomputers in 2017 had the average load of 92.3% and 88.7%, respectively [1]. Titan supercomputer in 2015 had at least 10% of its capacity unused [2].

One possible way to address this issue is adding low-priority jobs to be executed by the computational nodes that would otherwise remain idle. There are numerous problems in physics and other scientific fields that can produce a large number of jobs, e.g. series of Monte Carlo simulations with varying parameters. This approach was used, for example, to run ATLAS jobs on Titan supercomputer [2, 3].

* 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".

Conveniently, a lot of jobs that can be used to fill idle supercomputer nodes do not require parallel execution and thus can be started on an arbitrary number of idle nodes. However, filling all idle nodes will increase the average load at the cost of performance with respect to regular jobs: the idle nodes which can be used to execute additional low-priority jobs (filler jobs) would not otherwise necessarily remain idle for the duration of the execution, particularly if time requirements of the filler jobs are high. Usually it is not even possible to know in advance how long a node will remain idle, because newly submitted jobs or jobs completed ahead of time can change the schedule. So in order to reduce the negative impact of filler jobs on supercomputer performance with respect to regular jobs, filler jobs with relatively short execution time should be used.

We propose to run additional low-priority jobs in containers and use container migration tools to interrupt the job execution and resume it later, possibly on a different node. We describe a two-component system for managing and executing these jobs that can work alongside a supercomputer scheduler. We present the results of simulation of supercomputer workflow with and without the proposed system to demonstrate its potential advantages. We previously discussed this idea in [4]. A similar idea for utilizing idle resources of supercomputers was used in [5] where authors proposed a quasi scheduler to manage jobs of a special type (distinct from the type we use here).

2 Saving and Restoring Containers

Container virtualization is a method of isolating groups of processes and their environment, based on the Unix chroot mechanism. Containers are in many ways similar to virtual machines but they use the same OS kernel as the host machine. Several implementations of container virtualization offer tools for container migration. Some of the differences between the implementations of container migration are significant for our proposed use of containers, and there is some terminological difference as well. For simplicity, we will focus on Docker implementation of container migration.

In Docker [6], container migration is implemented as an experimental feature. It uses CRIU package (CRIU stands for Checkpoint/Restore in Userspace) allowing to freeze running applications and save them as collections of files. These collections of files, or checkpoints, can be later used to restore the application. The respective Docker feature can be used to create checkpoints for running containers, saving the processes running in these containers. Restoring the container from a checkpoint requires an identical container without any running processes. Our tests confirm that this experimental Docker feature works reliably and meets the requirements of our project: the checkpoints can be put to storage, restored on a different host machine, and a single container can be saved and restored multiple times consecutively.

We also conducted a series of tests to measure the time needed to start a container, create a checkpoint, and restart a container from a checkpoint. It is typical for container operations like creating a new container to take little

time. In our tests, starting a container took 1.25 s on average. However, creating a checkpoint involves copying to a permanent storage device all contents of a container, including RAM used by its processes, so we expected the creation time to grow approximately linearly with the size of the checkpoint. This was confirmed by our tests: creating a checkpoint for a container using 1 MB memory on average takes approximately 1 second, and at 200 MB memory and higher it takes approximately 5 seconds per 100 MB. It is more than 4 times slower than the sequential write speed of the HDD used in our tests (87 MB/s). Restoring containers from checkpoints also proved to be slow: over 1 second for a container using 1 MB memory and approximately 4 seconds per 100 MB for containers using over 200 MB memory. (By contrast, LXC [7] implementation of saving and restoring containers is several times faster, possibly because of their use of ZFS file system. However, there is a persistent error that prevents an LXC container from being saved and restored the second time after it was done once.)

Of course, these figures are not enough for an accurate estimate of the time required by checkpoint operations. Supercomputers can use better hardware, and write speed can be increased by the use of RAIDs. It is also possible that Docker developers will find a way to make checkpoint operations less time-consuming. On the other hand, the network of a supercomputer is a possible bottleneck, and if the computational nodes cannot access local storage of the other nodes then the shared storage is another one. Furthermore, utilizing multiple CPU cores of a single computational node requires sharing access to checkpoint storage between multiple single-core jobs. Therefore we can reasonably expect to lose, on average, at least several minutes of CPU time due to container operations and inactivity each time a computational node is used to run filler jobs.

For the purposes of the proposed system, several minutes is a significant delay. Because of this delay, we cannot let filler jobs run on a computational node until it is required for regular jobs: creating checkpoints for all filler jobs running on a node requires too much time. But there are several other possible approaches: for example, it is possible to periodically create checkpoints for running filler jobs without stopping the containers and stop the jobs on demand without creating new checkpoints and without losing too much progress. In this paper, we propose a different approach: to allot a time interval the node can run filler jobs and save the unfinished jobs at the end of the interval.

3 Managing Filler Jobs to Utilize Idle Supercomputer Nodes

We propose a two-component system to manage filler jobs and execute them inside containers on a supercomputer. The first component is an agent program. An instance of the program is launched on a computational node, creates or restores containers with filler jobs, and saves the containers before the allotted time is over. The second component is a control program that maintains the queue of filler jobs, stores information about job status, distributes filler jobs

between instances of the agent program, and interacts with the supercomputer scheduler.

Knowing the number of CPU cores per node, available memory, and the time it takes to start, checkpoint, and restore containers depending on the memory they use, it is possible to choose a minimum execution time for the agent program. However, an agent program instance can run longer than the minimum time, increasing the effective use of the node. This makes containerized filler jobs different from regular jobs: the scheduler does not have to abide by the requested execution time and can set its own execution time instead. In this, our filler jobs are similar to the jobs managed by the quasi scheduler proposed in [5] where the jobs do not have a fixed amount of nodes they need, instead allowing the quasi scheduler to determine the number of nodes they can use.

Thus we can choose an approach to setting the execution time for filler jobs. Based on tradeoffs between node utilization and performance with respect to regular jobs, we can find in advance an optimal execution time and use it for all filler jobs. However, this can result in filler jobs retaining nodes even when there are regular jobs that can use them. For example, suppose there is a regular job J waiting in the queue that requires 10 nodes. Another job terminates, releasing 2 nodes. Since 2 nodes is not enough to start J, they are given to filler jobs instead. Then 8 more nodes are released by other filler jobs, but it is not enough to start J so they are given to filler jobs once again. As a result, regular jobs can lose access to a significant number of nodes. Another idea is to set the termination time for filler jobs to match some other running job or jobs. However, user estimates of execution time are very inaccurate (see, e.g., [8]) so usually the termination time of a regular job cannot be predicted accurately. On the other hand, we control the termination time of filler jobs supervised by the agent program. Synchronizing the termination of agents can be as simple as having them all stop at the end of each hour (or any other period).

Now we can describe the components of the system to utilize idle supercomputer resources in more detail. Please note that both components of the system deal with filler jobs only. An instance of the agent program must:

- check the remaining time and stay idle if there is not enough time to start (or resume) a single job and then create a checkpoint for it, with some time left to actually run the job,
- request jobs from a control program,
- start or restart the jobs one by one inside containers; a job should only be started or restarted if this would leave enough time to create checkpoints for all running jobs,
- periodically check the running jobs and report the complete ones back to the control program,
- create checkpoints for the remaining jobs when there is just enough time left for it and report their status back to the control program,
- wait until the termination time and exit.

(The program needs the upper estimates of the time required to start or restart

a job and create a checkpoint as its parameters, as well as the minimal execution time for filler jobs.)

The control program must:

- submit jobs consisting of agent program instances to the supercomputer scheduler with the lowest priority and maintain their number according to the settings,
- maintain the queue of non-parallel low-priority jobs with records of their status,
- upon receiving a request from an agent, provide it with the jobs from the queue,
- track the jobs given to agents and update their status according to the information from the agents.

The supercomputer scheduler does not need any changes: filler jobs are submitted as regular (lowest-priority) jobs with the requested time equal to the interval between synchronized termination of the agents.

We implemented a prototype of the proposed system, working with Docker containers. Our tests show that it successfully launches filler jobs inside containers on idle computers and creates checkpoints for them before the allotted time is over. Filler jobs can be restored and saved multiple times on different nodes. We have not yet fully implemented a version of the agent program capable of running multiple filler jobs on a single node. Implementing and testing this part of the functionality will be a part of future work.

4 Simulation Results

Using a simulated job queue with a parameter distribution based on data from Lomonosov supercomputers, we performed a series of experiments to estimate the benefits of the proposed system. A detailed description of the experiments was given in [4], here we give a brief summary with some additional results.

The experiments consisted of two series. In the first series, the main scheduler queue was assumed to be full (always kept at 100 regular jobs), and in the second series, the regular jobs were added according to the Poisson distribution with the parameters chosen to approximate historical average load of Lomonosov supercomputers. The average load in the first series was high even with the basic scheduler only: 96.9% with 1024 nodes, 99.2% for 4000 nodes. We assumed that container and checkpoint operations cost 10 minutes every time a node runs filler jobs (if Docker checkpoint operations are used, this approximately corresponds to 8 jobs per node each using 1.5 GB memory on a hardware similar to that used in our tests). With this assumption, our system with one hour interval between synchronized agent termination increased the effective utilization to 99.0% with 1024 nodes and 99.5% with 4000 nodes. With a different type of job queue, the results were even better: the increase was from 95.5% to 98.9% with 1024 nodes and from 99.1% to 99.6% with 4000 nodes. For the first type of job queue, adding filler jobs decreased the average number of nodes running regular jobs (by 0.56%

with 1024 nodes and 0.19% with 4000 nodes). For the second type, the decrease was negligible.

In the experiments with the full queue, changing the interval between agent termination can sometimes improve the results but one hour is a good default choice. In the second series of experiments, one hour interval is too short: the effective resource utilization can be increased significantly by choosing 4 or even 6 hours. With 4 hours interval, the effective utilization is increased from 92.3% to 98.9% (first type of queue, 4000 nodes), and from 88.8% to 99.3% (second type of queue, 1500 nodes). There is no significant decrease in the average number of nodes running regular jobs.

In our additional experiments, we estimated the average waiting time for a regular job. Regular jobs were added according to the Poisson distribution. The waiting time for the first type of queue with 4000 nodes was 32 minutes with only regular jobs and 44 to 63 minutes with filler jobs depending on the interval between agent termination. For the second type of queue with 1500 nodes, the average waiting time was 89 minutes without filler jobs and 98 to 114 minutes with filler jobs. The termination interval for agent instances was between 0.5 and 4 hours. As these experiments show, adding filler jobs increases the waiting time for regular jobs even without reducing the number of nodes executing them. The additional delay is significant but relatively short given the increase in average load. By comparison, in the experiments where our system is not used and the average load is increased by adding regular jobs at a higher rate, the waiting time between 44 and 63 minutes approximately corresponds to the average load 93–95% for the first type of queue, and the waiting time between 98 and 114 minutes approximately corresponds to the average load 89–90% for the second type of the queue.

5 Conclusion

Our tests of container tools and the prototype demonstrate that the proposed system can work as intended. Docker containers are suitable for the purposes of the system, although in the current implementation the checkpoint operations are very slow. Our simulation experiments show that the system can increase the effective utilization of supercomputer resources, possibly by as much as 10%. The average effective utilization was increased in all our experiments. Possible downsides include the increased waiting time for regular jobs, and in some cases the decrease in the number of nodes running regular jobs. However, in many cases the decrease is not significant.

Acknowledgements. 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”. We would also like to thank Sergey Zhumatiy for the provided information and useful discussions.

References

1. Leonenkov, S., Zhumatiy, S.: Supercomputer Efficiency: Complex Approach Inspired by Lomonosov-2 History Evaluation. In: Russian Supercomputing Days: Proceedings of the International Conference (September 24–25, 2018, Moscow, Russia), pp. 518–528. Moscow State University (2018)
2. De, K. *et al.* [ATLAS Collaboration]: Integration of PanDA workload management system with Titan supercomputer at OLCF. Journal of Physics: Conference Series, **664**(9) p. 092020 (2015)
3. Barreiro Megino, F. *et al.* [ATLAS Collaboration]: Integration of Titan supercomputer at OLCF with ATLAS Production System. Journal of Physics: Conference Series, **898**(9) p. 092002 (2017)
4. Dubenskaya, J., Polyakov, S.: Improving the effective utilization of supercomputer resources by adding low-priority containerized jobs. In: Kryukov, A., Haungs, A. (eds.) 3rd International Workshop on Data Life Cycle in Physics 2019, CEUR Workshop Proceedings, vol. 2406, pp. 43–53. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen (2019)
5. Baranov, A.V., Kiselev, E.A., Lyakhovets, D.S.: The quasi scheduler for utilization of multiprocessing computing system's idle resources under control of the Management System of the Parallel Jobs. Bulletin of the South Ural State University. Series "Computational Mathematics and Software Engineering" **3**(4), 75–84 (2014) (in Russian)
6. https://docker.com/
7. https://linuxcontainers.org/
8. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling user runtime estimates. In: Feitelson, D.G. *et al.* (eds.) 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, vol. 3834, pp. 1–35. Springer-Verlag (2005)