Fair Resource Allocation for Running HPC Workloads Simultaneously

Ruslan Kuchumov(✉) and Vladimir Korkhov(✉)

Saint Petersburg State University,
7/9 Universitetskaya nab., St. Petersburg 199034, Russia
kuchumovri@gmail.com, v.korkhov@spbu.ru

Abstract. In high performance computing (HPC) job schedulers usually divide resources of computing nodes into slots. Each slot can be assigned to execute only a single job from the queue. In some cases, jobs do not fully utilize all available resources from the slot which leads to internal fragmentation, wasted resources and to an increase of queue wait time. In this paper, we propose fair resource allocation strategies that can be applied in job schedulers for resource allocation. We cover such resources as CPU time, residential memory and network bandwidth.

Keywords: High performance computing · Scheduling · Fair resource allocation

1 Introduction

In high performance computing (HPC) field job schedulers are widely used for maintaining queues of jobs created by different users and assigning nodes of the computing cluster to execute these jobs. Usually, at any moment one computing node is assigned for execution of a single job. Sometimes, nodes are divided into slots (for example, one slot per CPU core) and a slot is assigned to a single job. It leads to better utilization of computing nodes and to an increase of throughput of a cluster, as multiple jobs can be executed simultaneously.

But nevertheless, with this approach some computing nodes or slots may not be fully utilized. For example, CPU time of a slot may not be consumed fully, when the jobs it is executing are io- or network-intensive. Underutilized slots are not schedulable, they can not be reclaimed by the scheduler and the resources they have occupied become idle for some portion of job make-span. Right next to an under-utilized slot there may be an over-utilized one that can benefit from these idle resources.

The wasted resources of a node or a slot in some cases may be significant and expensive. For example, when performing HPC in a public cloud, it would lead to greater prices of job computations as users are charged for these idle resources as well. Thus, using the strategy of assigning a single slot to a job is not ideal.

By assigning a single computing node to execute multiple jobs it would be possible to decrease the amounts of idle resources compared to aforementioned

S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 740–751, 2019.
https://doi.org/10.1007/978-3-030-24305-0_55

strategy with slots. Furthermore, more jobs can be executed at the same time as node resources may be utilized more efficiently. All that would lead to decrease of job wait time in the queue and higher throughput of the cluster.

In order to achieve that, it would not be enough to execute multiple jobs simultaneously, as fair resource allocation between jobs must be provided. For example, one job may spawn more processes than the other, so on average it would receive more CPU time as the scheduler of operating system distributes it evenly.

In this paper we continue our earlier research [1–3] and cover fair resource allocation strategies for the resources such as CPU time, main memory and network bandwidth. These strategies guarantee jobs an equal shares of each resource and also allow jobs to exceed their shares at the expense of under-utilized shares of other jobs (which can later be reclaimed back).

2 CPU Time

Linux kernel scheduler by default uses Completely Fair Scheduling (CFS) strategy to equally distribute CPU time among threads and processes [4]. By default it treats all the tasks (e.g. threads and processes) the same way without considering their position in hierarchy. Because of that, when one job creates more child processes than the other, it would get on average more CPU time.

CFS scheduling strategy is based on the concept of virtual time. Each scheduler task in the queue has virtual time which equals to the weighted CPU time the task has spent executing on CPU. It is weighted so that the more the weight that task has, the slower its virtual time flows.

Runqueues are represented by red-black tree sorted by virtual time. It allows to quickly find the next task which would be executed on the CPU. CFS scheduler always picks the task with lowest virtual time and executes it for certain period of time without preemption. This time period (called slice) equals to the scheduling period proportionally divided between all the weights of tasks in the queue.

Each logical CPU core has its own runqueue. Each CPU is included in a scheduling group, which are, in turn, included in scheduling domains. Scheduling groups form non-intersecting sets of CPUs. Task balancing within a domain occurs between the groups. Each group has a load value which is defined as load of all the runqueues it includes, and when these loads are becoming out of balance the tasks are migrated. Scheduling domains may have hierarchical structure. The scheduler traverses this structure and periodically performs the same balancing procedure as for groups.

CFS scheduling strategy is also hierarchical. Each task is represented by a scheduling entity which can either be a leaf task (process, thread or tasklet) or a task group with another CFS queue. In case of non-leaf runqueue, the CPU time would be equally distributed between both tasks and task groups.

By creating control groups it is possible to define a hierarchy inside of CFS scheduler. In case we create a control group for each job, nest it in the same parent control group and assign equal weights to each cgroup we would be able

to distribute CPU time equally, at first, between job and then between processes within an job.

To demonstrate that, we have placated two CPU-intensive benchmarks into two separate control groups. The benchmark instance in the first control group spawn $N = 4$ CPU-intensive threads and the time the instance in the second control group spawns $M = 2$ threads. In our test-bed we had 4 CPU cores, which yield $C = 4$ units of CPU-time.

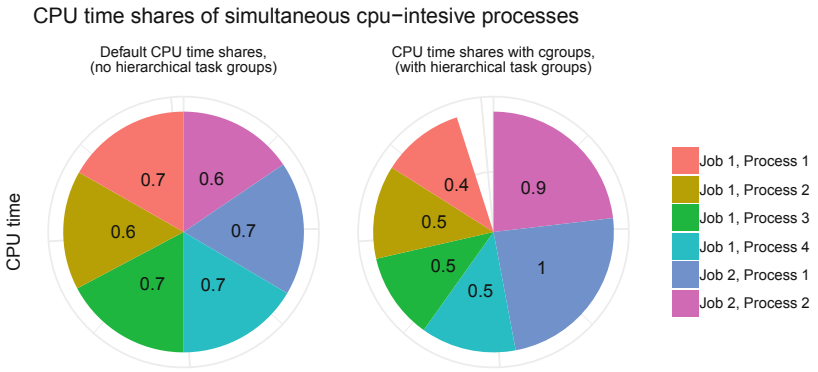

Fig. 1. Comparison of CPU shares with and without task groups. Left chart shows equal CPU time distribution between jobs processes. After placing jobs into separate control groups, CPU time was distributed fairly, at first, between groups and then between processes within groups.

Figure 1 shows that without placing these benchmark instances into control group, reported CPU-time was distributed evenly and was equal to $0.6 \approx C/(N + M)$ for each thread. After defining control groups, CPU time was distributed evenly between the groups and after that within a group. CPU-time ratio in this case was equal to $0.5 \approx C/2N$ and $1 \approx C/2M$ for the first and the second group respectively.

3 Memory

For equally distributing memory between jobs we had also used control groups since it is the standard way of limiting the usage of both swap and residential memory.

At any point in time, given the number of running jobs we may equally divide all available memory among them, but the problems would arise when one of the memory of one of control groups would grow and would reach its limit. There are several possible scenarios:

– Other groups have spare memory. In this case, it is better to rent their space instead of, for example, using swap or freezing the group. But, we have to

guarantee that, when these groups start to grow, they would have higher priority and would be able to reclaim memory pages back.
– Other groups do not have spare memory. In this case, more radical actions should be taken, since the system is running out of memory. Some unused memory pages can be migrated from residential memory to the swap without affecting the job. It would allow to shrink residential memory consumption. After the point when shrinking residential memory shares is no longer possible, a victim job must be chosen that would be almost completely displaced to swap or frozen.

3.1 Using Memory Pages from Other Groups

It is possible to guarantee the amount of residential memory that would not be reclaimed by other groups by setting the low boundary ('memory.low') limit. If we divide the total available memory by the number of groups and set it as the low boundary, then when this share is underutilized by job, all of its pages would stay in memory. At the same time, others may claim the free space. Even when the job starts to reclaim pages back (below the low limit), the pages would stay in memory, while the pages of other job may be displaced to the swap.

For example, two jobs have low limit of 100 MB. We had limited the total amount of residential memory available to both jobs to 200 MB by setting root's control group high and max memory limit to this value. The first one (job1) grows by 20 MB regularly, while the other one (job2) takes 75 MB constantly.

As the result (Fig. 2), job2 always stays in memory (at 75 MB), job1 reclaimed some of the space from job1 ($100 - 75 = 25$ MB), and when this space also becomes full (because of the root limit of 200 MB) it starts to use swap.

3.2 Reclaiming Memory Pages

The next case is when underutilized group starts to reclaim its space and becomes over-utilized. The group that was using the space of the fist group must now "find" another space to claim.

In the next example both groups have low limit set to 100 MB and root's high limit is 200 MB again. The first group (job1) constantly uses 160 MB, while the second group (job2) periodically grows to 200 MB by 20 MB. The first group is over-utilized and take more space in RES memory than its low limit; the second group starts to grow, reclaims its pages, and becomes over-utilized as well.

As the result (Fig. 3), the pages of the first group has moved to swap when the pages from the second group has reclaimed their space. When both groups have reached their limit of 100 MB and the second group has continued to grow, it started to use swap.

Shrinking Residential Memory Usage. When the memory is distributed tightly, and some tasks start to grow, it is necessary to move some page to swap. Memory stall information that is introduced in recent kernel patches (pressure

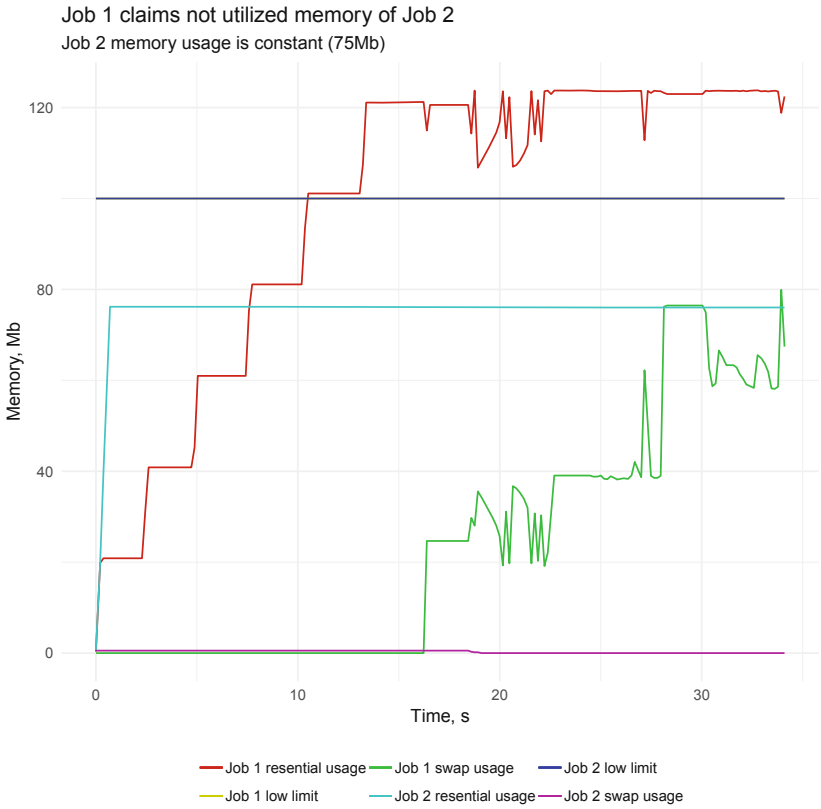

Fig. 2. Memory distribution between two jobs within confined space (200 Mb). Each job has a low memory limit of 100 Mb. Memory usage of Job 2 is constant (75 Mb) and memory usage of Job 1 grows by 20 Mb every 2 s. It can be noticed, that Job 1 claims unused memory of job 1 after 10 s and after that it starts to use swap.

stall information, PSI) may help to choose the victim group. By migrating unused pages it is possible to shrink residential memory usage causing the minimal effect to the job.

Recent upstream patches to the Linux Kernel (4.20) introduce pressure stall information (PSI) per control group. For the memory resource it presents the amount of time each task in control group has spent performing memory operations. Among these operations are, for example, memory page locking and waiting on the lock, page disk writebacks, memory compaction (by kcompactd and before page allocations). This time is averaged among all CPUs runqueus and weighed by non-idle runqueue time. Similar information is defined for disk I/O, CPU time.

In our case, PSI value would allow to tell by how much the job is affected by moving its pages to the swap. In case when the pages that are accessed frequently are required to move to the swap, memory thrashing would begin and PSI values

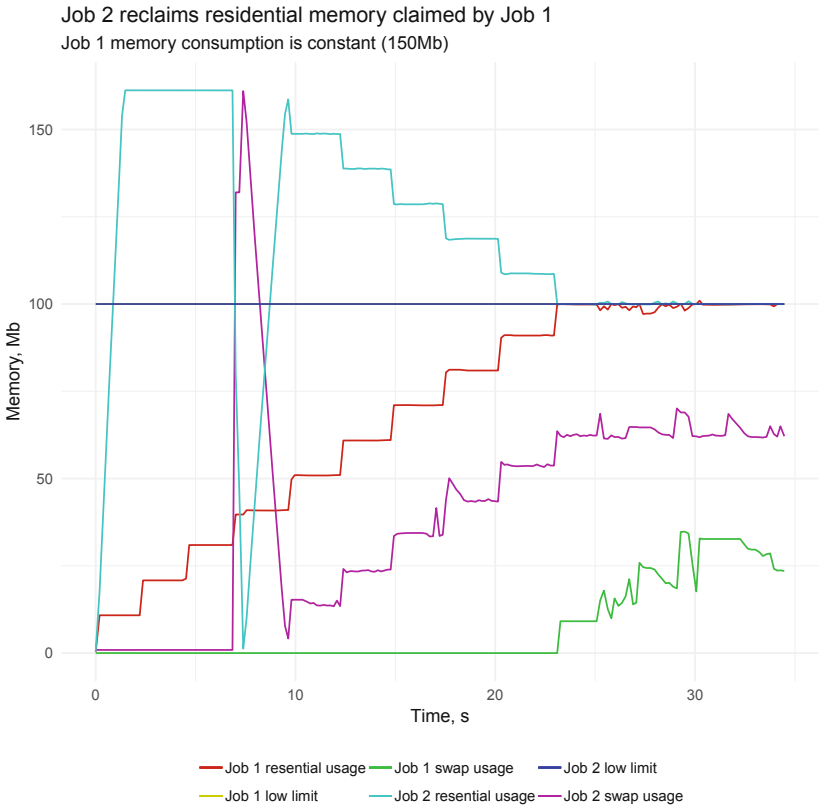

Fig. 3. Memory distribution between two jobs within confined space (200 Mb). Each job has a low memory limit of 100 Mb. Memory usage of Job 2 is constant (150 Mb) and it has partially claimed the share of Job 1 (by 50 Mb). Memory usage of Job 1 grows by 20 Mb every 2 s. It can be noticed, that Job 1 gradually reclaims its memory shares causing memory pages of Job 2 to migrate to swap. At 22nd second both jobs reach their low limits, and after this point Job 2 has no other space to grow besides swap.

would rise. So, by gradually moving pages to swap until PSI value starts to rise, it is possible to shrink group without interfering with job.

In the following example (Fig. 4), the first job (job1) requires 200 MB, but its low limit is restricted to 100 MB. As the time goes by, low boundary is gradually decreased causing page migration to the swap. Similar to the previous experiments, the second job (not shown) constantly uses the rest of the memory (100 MB). Nominal and real memory shares show specified and observed residential memory share ($RES/(RES+SWAP)$) of the each job. The third line in the plot shows pressure of the first job. The ratio of unused pages is controlled by stride (1 Kb, 8 Kb and 12 Kb) parameter which means that every 'stride' byte

of allocated memory is accessed, so when 'stride' is 12 Kb, every third memory page is touched (considering pagesize is 4 Kb).

Figure 4 shows that PSI value increases when more and more pages migrate to the swap. Also, when pages are unused (large stride value), PSI value becomes smaller. As the result, PSI value allows to measure how much the job is affected by moving its pages to the swap.

4 Network Bandwidth

By default network bandwidth is shared evenly between connection sessions. Some jobs may open multiple sessions, which would lead to uneven bandwidth shares (Fig. 5).

Linux kernel allows to define queuing disciplines that can shape and schedule network traffic before it would enter network device queue. Multiple different queuing disciplines can be applied to define hierarchical queue structure and provide non-trivial QoS policies.

Stochastic Fairness Queuing (SFQ) is a classless queuing discipline which allows to schedule network traffic fairly between traffic flows. It is based on the concept of hash buckets (or the flows). When the network packet is enqueued a hash function determines the bucket it would belong to based on the given key. When the next packet needs to be dequeued it is selected from the buckets in round-robin fashion.

The key of the packet that would be used in a hash function can be assigned using flow filer classifiers. As our goal there is to evenly distribute network traffic between jobs, the packets coming from the sessions within a job should be scheduled to the same bucket and packets coming from different jobs should be scheduled to different buckets.

Since we have already used control groups for fair CPU time and memory allocations it is worth to continue using them for networking as well. Linux kernel firewall (iptables) allows to define netfilter rules to mark packets that are coming from the threads and processes within a specific control groups. By using Linux traffic control flow classifier to define flows based on the packet mark, or, in the other words, to use packet mark as key in bucket hash function, it is possible to schedule multiple flows coming from the same job into a single hash bucket. This would allow us to evenly distribute bandwidth among different jobs.

Since SFQ does not shape the traffic, its effect would be noticeable only when the maximum throughput of the data link is reached. In our case, it would mean that one job may take all available bandwidth when the other jobs do not require it, but when all jobs require network access, it would be shared evenly.

In our experiments demonstrating the feasibility of this concept we have used two hosts with Gigabit Ethernet NIC connected together with a patch cord. We had used iperf benchmark to measure interconnect speed in different conditions. iperf benchmark has reported us a baseline bandwidth of 900 Mbit/s with a single running client and no other processes using these NICs.

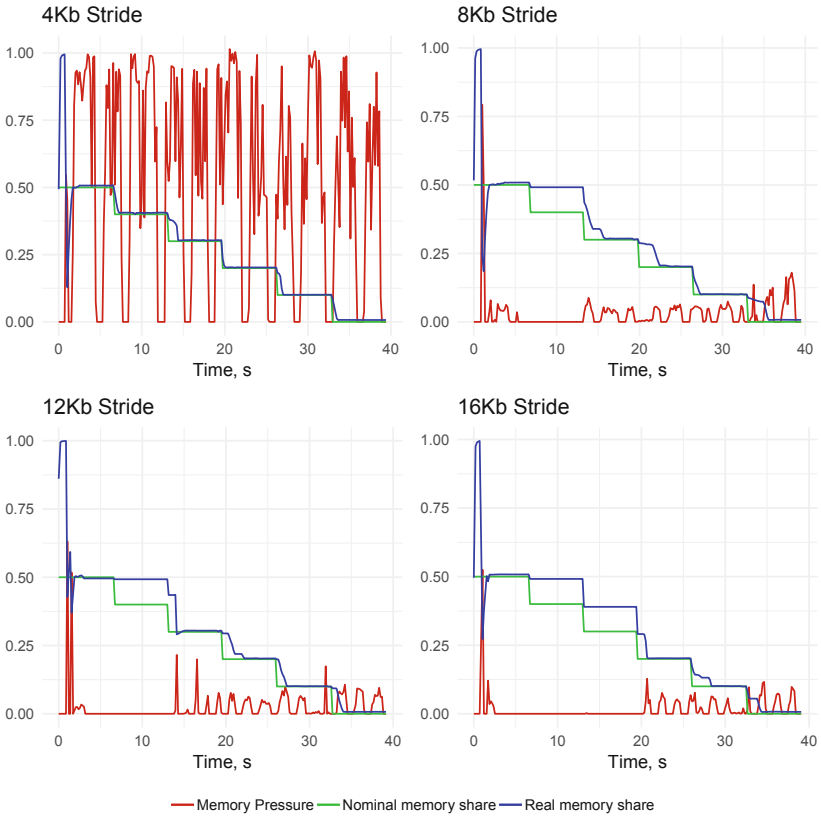

Fig. 4. Memory pressure value for different shares of residential and swap memory and for different page access patterns. It can be noticed that, when every page is accessed (4 Kb stride) the memory pressure is high regardless the number of pages in residential memory. When the number of accessed pages is significantly lower, the pressure decreases. More pages can be moved to swap without causing any noticeable changes in pressure, for example, when the stride is 16 Kb, memory pressure starts to rise only when 25% of pages are left in the residential memory.

To demonstrate even bandwidth distribution between connection session with default queueing policy (codel) we had started 3 iperf client instances simultaneously. As the result, bandwidth of 314, 312 and 380 Mbit/s was reported, which is approximately equal to the third of a baseline bandwidth.

To simulate jobs with uneven number of network connection, we had placed three iperf instances in two control groups. As the result, the first control group had a single connection session, while the second control group had two connection session. After creating SFQ queuing policy and flow specifier for the traffic coming from these control groups, the first connection should get 1/2 and the second and the third should get 1/4 of the baseline bandwidth.

We had created two sibling control group ‘d1’ and ‘d2’. The following ‘iptables’ rules allow to mark IP packets coming from the processes inside of these cgroups with ‘11’ and ‘12’ tags.

```
iptables -A POSTROUTING -t mangle -m cgroup --path d1 \
    -j MARK --set-mark 11
iptables -A POSTROUTING -t mangle -m cgroup --path d2 \
    -j MARK --set-mark 12
```

After that we had created root SFQ queue and a flow filter that would apply a hash function over packet tags (‘hash keys mark’). ‘perturb 2’ allows to rotate bucket hash function every 2 s. ‘divisor 1024’ specifies key modulo or the number of buckets.

```
tc qdisc add dev eth0 root handle 1: sfq perturb 2
tc filter add dev eth0 parent 1: protocol ip handle 1 \
    flow hash keys mark divisor 1024
```

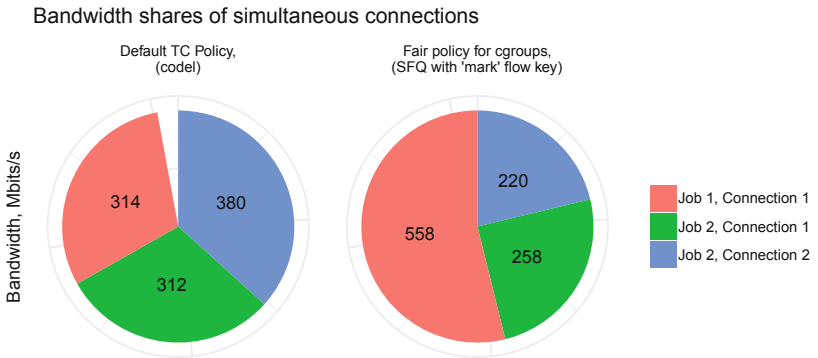

Fig. 5. Comparison of network bandwidth shares with and without fair policy. Left chart shows equal bandwidth distribution between connection sessions. After placing jobs into separate control groups, marking groups outgoing traffic and creating flow classifier for packets marks, the network bandwidth was distributed fairly, at first, between control groups and then between connection sessions within groups.

As the result (Fig. 5), the bandwidth is shared evenly across two jobs, as expected. The first job with a single connection received a half of a bandwidth (578 Mbits/s), and the second job received another half of a bandwidth for two connections (258 and 220 Mbits/s).

5 Related Works

Some batch schedulers such as SLURM [5] and Son of a Grid Engine [6] allow to oversubscribe CPU cores in some way. They achieve that by allowing administrators to specify load threshold under which oversubscription may occur. The fairness of CPU time distribution between jobs is left out of scope in these cases.

On the other hand, the majority of schedulers allow to launch multiple jobs in a single node at the same time by dividing them between CPU cores. Since the total amount of memory required by all of the jobs running simultaneously may exceed all available memory of the node, memory over-subscription may occur. In order to handle this situation, some schedulers, e.g. Maui [7], offload some of the pages to the swap memory up to the specified over-commit factor. Another strategy utilized by LSF [8] scheduler, where jobs can be preempted or suspended to free residential memory for higher priority jobs. Similar to CPU time resource, the fairness of memory distribution is not covered in these solutions.

The problems of unfair resource allocation of HPC schedulers were covered from both technological and users perspectives in [9] and in the works linked therein.

For CPU time distribution the paper [10] states the problem of unfair CPU time distribution in the Linux Kernel scheduler. It is caused by the processes that have more child processes than the others. The proposed solution in this paper is scheduler modification.

The usage of Linux control groups for enforcing the limits on residential memory shares is a commonly known practice. Similar approaches for migration of pages to swap based on memory pressure values are used for virtual machines [11]. For virtual machines, memory pressure value is provided by hypervisors and it is specific to it. But, in this paper, to achieve our goals we are using standard capabilities provided by the recent updates in the Linux Kernel, such as process stall information (PSI).

6 Conclusion and Future Work

In this paper we have proposed resource allocation strategies that can be applied to HPC workloads. By grouping resource consumers into separate control groups it is possible to provide fair resource shares between groups regardless on the number of resource consumers in each group.

We have covered the resources such as CPU time, memory and network bandwidth. These strategies guarantee jobs equal shares of each resource and also allow jobs to exceed their shares at the expense of under-utilized shares of other jobs (which can later be reclaimed back).

CPU time is shared evenly by using abilities of Linux Kernel CFS scheduler, which allows to create hierarchical structure of processes and the groups of processes. CPU time is distributed evenly between groups and processes that belong to the same parent. By creating a task group (control group) for each job it is possible to distribute CPU time evenly.

To control memory page allocations we have used control groups, which allow to define guaranteed limit of residential memory for each job. When a job exceeds this limit, it can utilize spare memory of other jobs that have not reached it. Later, when these under-utilized jobs start to grow, they can reclaim their pages. We have also showed that it is possible to migrate unused memory pages to swap without causing any effect by using memory pressure as a feedback.

Network bandwidth is also shared fairly between control groups by using SFQ traffic control queuing policy. Each packet that is coming from connections within a control group is marked and then this mark is used to map traffic flows within SFQ queue. SFQ allows us to schedule traffic of multiple flows and to achieve fair distribution regardless the number of sessions within each group.

HPC workloads have a high demand on these resources and they usually define the cost of the computational cluster. Job schedulers that are commonly used right now for HPC workloads simply divide available resources of a computing node between jobs and do not allow any job to exceed available resource shares. This strategy leads to fragmentation problems and, in turn, to wasted resources.

When moving HPC workloads to the cloud, where users are billed for the resources they use, the cost of this fragmentation may be significant. In this case, using HPC schedulers would not be the best solution. Moreover, some workloads may have different resource demands, e.g. some jobs can be cpu-intensive, some network-intensive and others may require only accelerators, since each node may have all of these resources, these jobs can be executed simultaneously without interference.

In the future, based on the results of this paper, we are going to propose a different kind of an HPC scheduler, which would not be based on discrete slots but rather on continuous resource shares. This, in turn, would allow us not only to decrease resource fragmentation, but also to decrease job wait time in the queue.

References

1. Kuchumov, R., Petrunin, V., Korkhov, V., Balashov, N., Kutovskiy, N., Sokolov, I.: Design and implementation of a service for cloud HPC computations. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10963, pp. 103–112. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95171-3_9
2. Kuchumov, R.I., Korkhov, V.V.: Design and implementation of a service for performing HPC computations in cloud environment. In: CEUR Workshop Proceedings, vol. 2267, pp. 233–236 (2018)
3. Korkhov, V., Kobyshev, S., Degtyarev, A., Bogdanov, A.: Light-weight cloud-based virtual computing infrastructure for distributed applications and hadoop clusters. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10408, pp. 399–411. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62404-4_29
4. Wong, C.S., Tan, I.K.T., Kumari, R.D., Lam, J.W., Fun, W.: Fairness and interactive performance of o(1) and CFS Linux kernel schedulers. In: International Symposium on Information Technology, vol. 4, pp. 1–8. IEEE, August 2008
5. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple Linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). https://doi.org/10.1007/10968987_3
6. SGE, Son of Grid Engine. https://arc.liv.ac.uk/trac/SGE. Accessed 07 May 2019
7. Maui: 5.2 Node Allocation, Page Redirection. http://docs.adaptivecomputing.com/maui/5.2nodeallocation.php. Accessed 07 May 2019

8. LSF: Fairshare Scheduling. https://www.bsc.es/support/LSF/9.1.2/lsf_admin/index.htm?chap_fairshare_lsf_admin.html~main. Accessed 07 May 2019
9. Sedighi, A., Deng, Y., Zhang, P.: Fariness of task scheduling in high performance computing environments. Scalable Comput. Pract. Experience **15**(3), 271–288 (2014)
10. Wong, C.S., Tan, I., Kumari, R.D., Wey, F.: Towards achieving fairness in the Linux scheduler. ACM SIGOPS Operat. Syst. Rev. **42**(5), 34–43 (2008)
11. Waldspurger, C.A.: Memory resource management in VMware ESX server. ACM SIGOPS Oper. Syst. Rev. **36**(SI), 181–194 (2002)