Improving HPC applications scheduling with predictions based on automatically-collected historical data

Carlos Fenoy García

A thesis presented for the degree of
Master CANS

Advisor: Julita Corbalán González

Department of Computer Architecture
Universitat Politècnica de Catalunya

August 2014

Abstract

This work analyses the benefits of a system in an HPC environment, which being able to get real performance data from jobs, uses it for future scheduling in order to improve the performance of the applications with minimal user input. The study is focused on the memory bandwidth usage of applications and its impact on the running time when sharing the same node with other jobs. The data used for scheduling purposes is extracted from the hardware counters during the application execution, identified by a tag specified by the user. This information allows the system to predict the resource requirements for a job and allocate it more effectively.

The study shows that combining the automatic performance monitoring data collection and a scheduling aware of limited resources -memory bandwidth- better results can be achieved rather than using standard scheduling strategies and without forcing the users to provide more information about their jobs that they may not completely know.

Contents

Chapter 1

Introduction

1.1 Introduction

Since the beginning of this century, supercomputers have rapidly increased their sizes. The evolution of the interconnection technologies has allowed a significant increase in the number of compute nodes forming a supercomputer: from the ASCI White of 2000 with 512 nodes and 8192 processing units to modern Blue Gene machines, with almost 100,000 nodes and more than 1,500,000 processor cores. Obviously, the amount of memory available per core has also increased from roughly a gigabyte per core of the ASCI White to 4, 8 or even more in nowadays supercomputers, without considering shared memory machines which have considerably larger amounts of memory per core.

The amount of resources has increased but also has the performance of the different elements conforming a computer. Today faster processors, memories or disks can be found, but not all the components have evolved in the same way, and the difference between the performance of the different parts is becoming larger. Cpus are almost doubling their frequency every year, while hard disks suffer from low improvement in terms of speed, and even though memory has also evolved, its evolving rate is still slower than cpu's. This difference in the pace of improvement is what can reduce the benefits of having higher cpus frequencies if accompanied by very slow disks or memories, as there will not be enough work for the cpu to process.

In order to manage all the resources available on such large machines, supercomputers usually use resource management applications that take care of keeping records of the available and allocated resources at any time, for instance, to name two of this applications, Slurm and Torque. In conjunction with the resource manager, batch schedulers systems are usually found, responsible for managing the users jobs according to the administrators policies and the availability of resources. These two pieces of software can also be found integrated in one application, like the late versions of Slurm, Load Sharing Facility (LSF) or Sun Grid Engine (SGE), among many others. These applications are very complex systems that must be aware of all the details of the system in order to

carry out their duty properly.

Due to the complexity of the management of very large machines, resource managers have historically focused their efforts on managing cpus and memory, although some advanced versions also manage the amount of local disk space on every compute node, and even more recently, the availability of coprocessors like GPUs or the Xeon Phi. However, there are some resources that can not be directly managed, like the network bandwidth available in a compute node or the memory bandwidth available to a cpu.

Monitoring the usage of the resources has also been a point of interest in the supercomputing environment. The necessity to understand the behaviour of the applications and the systems on top of which these are running has led to the development of several tools that allow to get a very detailed view of the usage of every little part of a computer. Several solutions exist for active and passive job monitoring: from general tools for server monitoring, like Ganglia, to tracing tools that can give the maximum level of details, like Extrae. There are several other tools that cover the range between both ends, like PerfMiner. The use of hardware performance counters included in all modern processors enable applications to obtain information of the number of cpu cycles that were used or the number of branch predictions that failed during the execution, among other data.

The growth of the resources per node in supercomputers has caused the increase of node sharing between jobs of different users. This policy has raised the criticality of the resource management of the compute nodes. Current resource managers succeed in handling standard resources, cpus and memory, but fail to ensure the expected performance due to the lack of control of non common resources, like the amount of IO bandwidth or the memory bandwidth.

It is not always easy or possible to measure the amount of these resources available in a node, and is even more difficult to know in advance the usage of the resources required by the applications. Users usually do not know how their applications behave and so it is not reasonable to ask them to state the amount of non traditional resources needed by their jobs. Even if the users had an idea of the resources they may be consuming, current solutions do not prevent them to understate the amount of resources required in order to gain advantage bypassing the policies. The current monitoring systems could be of help, but there is no connection between the data collected by them and the scheduling of applications.

Due to the impact of shared resources in the applications performance and the lack of trusty mechanisms to ensure the right behaviour of the users in the resource requirements, the need for a new procedure to simplify the resource requests arises. A possible solution for this procedure may come from the combined use of the management and scheduler applications and the data collected by the monitoring system.

1.2 Objectives and results evaluation

The main goal of this work is to create and analyse a system to fill in the current gap between the information provided by the user and the information really needed by the scheduler to improve the overall performance of the system. Using existing software and previous studies and work, a system will be generated to improve the allocation of jobs based on real performance data gathered during the execution of the users applications. This information will allow the system to predict the amount of resources a job needs without asking the user for it.

This new system will be compared with the default Slurm policy and with a previous work that also considers the existence of other limiting resources- the memory-bandwidth-. Several workloads will be generated and executed, and Paraver traces will be extracted from these executions to analyse the behaviour of the different policies and their differences.

Chapter 2

Background

2.1 Related Work

This study is based on different existing studies and applications concerning jobs scheduling and monitoring of high performance computing jobs.

Job scheduling have been a matter of research due to its impact on HPC performance and cluster utilisation. How and by which factors jobs are scheduled are taken into account when setting up a scheduling system, as well as how the performance of this jobs is monitored. Both fields, job scheduling and monitoring, have been largely studied and implemented in different applications, whose results are the basis for the present study.

Job scheduling based on non conventional restricted resources, specifically, the memory bandwidth on the compute nodes, was described by Guim et al. (2008) in the work "Job-guided scheduling strategies for multi-site HPC infrastructures" [3]. This work describes a possible solution for memory bandwidth-bounded jobs and its allocation on distributed systems. This work was afterwards adapted to Slurm [4] and analysed by Fenoy (2010) [2]. This adaptation showed that the introduction of new resources into conventional schedulers can help in the prevention of resources waste caused by jobs dying after reaching their time limit without having ended their amount of work. These new resources can also help in the reduction of jobs execution times, that can have a direct effect in time limited tasks, eg. weather forecasting.

The automatic performance monitoring collection was studied and developed by Mucci et al (2005) in their PerfMiner[5] application. This system transparently monitors several hardware performance counters during the jobs executions and stores them in a database for later analysis of the applications performance. This work was adapted to Slurm by Abellan (2008) [1], and used in production in MareNostrum 2 supercomputer at the BSC-CNS monitoring all the clusters applications and enabling the users to understand the behaviour of their applications.

2.2 Environment

The present study was developed using the MareNostrum 3 supercomputer at the BSC- CNS, an IBM iDataplex dx360 m4 cluster consisting of 3056 compute nodes each of those with 2 sockets SandyBridge E5-2670 cpus with 8 cores, 32GB of RAM and Infiniband FDR10 40Gb/s for the compute nodes interconnection. MareNostrum uses GPFS as a cluster filesystem and LSF for job scheduling.

On top of the standard MareNostrum installation, for the purposes of this study PapiEx, PerfMiner and Slurm were installed.

PapiEx is an application using PAPI, a library for accessing the hardware performance counters, that preloads a library to automatically obtain performance measurements of the applications. PapiEx allows to select which performance counters should be collected and is able to monitor threads and processes creations reporting also the children performance. All the information is stored in text files with process and threads identification in the file names.

All the information generated by PapiEx is processed by PerfMiner and inserted in a MySQL database. For each job a record is generated in the database. The processes related with this job are later inserted to the database and for each process the threads are also included. Once all the threads are inserted, PerfMiner automatically creates a table for each performance counter not seen before and creates a record for each thread in each counter table. This way all the processes information is efficiently stored and allows the user to get the desired information without having to filter out lots of information.

Slurm is the resource manager and scheduler chosen for the implementation of the scheduling policies studied in this work. The main reason for selecting Slurm over other possible candidates is the availability of the source code and the existence of previous implementations of use for the present study on this resource manager (see "Related work"). Although it would have been interesting to implement the policies of this work in the main scheduler of MareNostrum, LSF, the closed source model of this software has made this option unfeasible. In addition to these considerations, Slurm is a widespread software used in several of the top 500 supercomputers.

Due to the impossibility to measure the exact amount of memory bandwidth consumed by a process, because of the existence of several factors affecting it (eg. the hardware prefetcher, the memory writes), the study is focused on the measurement of the total cache misses in the last level cache (LLC_TCM), available through the PAPI interface whereas the other contributors to the memory bandwidth are not easily available.

Slurm manages, as standard resources, cores and memory available on every node. In order to use it for the tests, the policy that takes into account the memory-bandwidth resource for scheduling -Less Consume- (see "related work") is used adapted to a newer version of Slurm (2.5.0) than the one originally used. This policy enables the users to specify the amount of memory bandwidth required by a job and avoid an overallocation of the memory bandwidth, which will result in a decrease of the overall performance.

On top of this modifications, a job submit plug-in for Slurm combining the Less Consume policy and the monitoring data was developed -Historical performance (HPerf)-. This plug-in enables the user to use a "tag" to identify a job instead of specifying the amount of memory bandwidth required. Given a tag, the plug-in will look for it in the database and extract the average amount of last-level total cache misses (LLC_TCM) for the jobs marked with this tag. If no previous record is found, the plug-in will force the job to run on exclusively allocated nodes in order to obtain a first measure of the LLC_TCM. If there is any record on the database, the average value of all the records will be used as the "memory-bandwidth" value for this job.

The same Slurm version (v. 2.5.0) is used throughout the study.

Chapter 3

Tests

3.1 Preparation

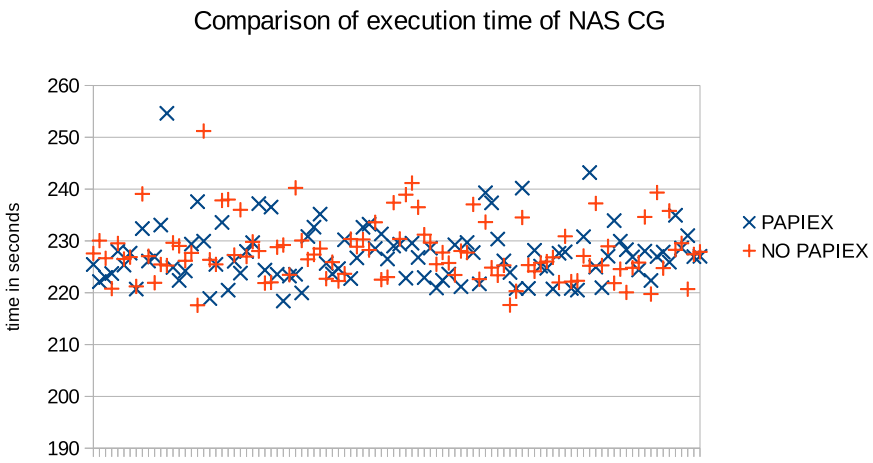


Figure 3.1: Comparison between executions with and without PapiEx

As PerfMiner would be used, which uses PapiEx underneath, a study of the overhead added by PapiEx was performed. A series of a hundred independent executions of CG application from the NAS-PB benchmark suite [6] were performed in the same series of nodes with and without using PapiEx. As can be seen in figures 3.1 and 3.2, the overhead added by PapiEx for an execution of CG is negligible, as there is almost no difference between both series of executions.

For testing and comparing the benefits of the proposed solution (HPerf) with the standard Slurm allocation algorithm, Consumable Resources (ConsRes), and Less Consume (LessC), a series of workloads were generated.

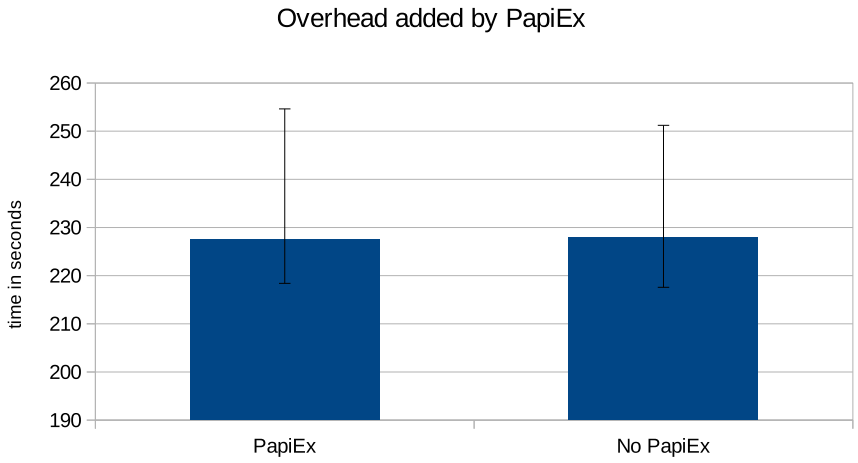


Figure 3.2: Execution time of executions with and without PapiEx

For the generation of the workload a previous study and characterization of applications was performed in order to select which applications would be run finally in the tests. The applications studied are the CG application from the NAS-PB and a synthetic test application which maximises its memory bandwidth usage. For the CG application the classes B, C and D were studied using 4 to 64 cpus. Three categories were created depending on the LLC_TCM values of the executions of the tests, being the CG class D and synthetic test assigned to the "high" category, CG class C to the "medium" category and finally CG class B to the "low" category.

A previous study of the impact of the distribution of tasks in the performance of the executions and its execution time was done to acquire the knowledge to understand if the final results of the project made sense. For this study of the distribution impact, several jobs were run using PapiEx with different numbers of iterations and number of processes of the different classes of the CG test, in order to obtain measurements of the LLC_TCM and the execution time.

Figure 3.3 shows the effect of the distribution in the execution time of the CG with 64 processors. As can be seen in the image the performance of the test improve from 16 processes per node (ppn) in 4 nodes to 8 processes per node in 8 nodes. There is also an improvement between 8 ppn and 4 ppn, resulting this last execution the better configuration. From 4 ppn to 2 ppn, the benefits of having more memory bandwidth available per process blur due to the overhead of the communications between higher number of nodes. The longer the time of the execution the higher are the probabilities of suffering from system noise effects as can be seen in the execution with 16 ppn at high values of iterations, where the execution time varies from the expected value.

At the same time than the analysis of the impact of execution time depending on the distribution of tasks, measurements of the LLC_TCM of each of the processes run were obtained to establish the amount of resources needed per

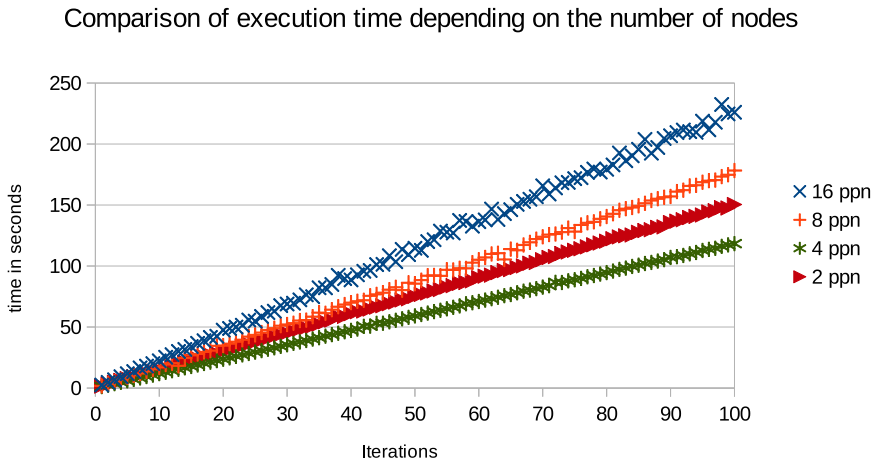


Figure 3.3: Execution time by number of iterations with different distributions

each class and size.

Having performed all the test for the candidate applications, table 3.1 shows the final list of applications used for the workload generation with their memory bandwidth usage and classified according to the amount of memory bandwidth required grouped in the three categories: high (h), medium (m) and low (l). Due to the amount of memory required by CG class D, low process values of this application were replaced by the synthetic test that requires significantly lower amount of memory. This way adding more complexity to the workload is prevented by avoiding the management of the memory requirements per job.

3.2 Workload Generation

After having classified all the target applications, the Lublin and Feitelson model was used to generate a workload. This model allows the settings of different parameters to specify the size of the cluster, the minimum and maximum amount of cores a job can use, the minimum and maximum duration of the jobs, among others. A workload with 100 jobs was generated using this model, containing the basic information of the jobs (size, duration and arrival time).

Once the workload was generated by the model, two final workloads were created from it: one with most of the jobs in the "high" category (Workload "High") and another one with half of the jobs on the "high" and the other half distributed between the "medium" and "low" categories (Workload "Medium"). Each of the jobs described in the workloads is transformed into a Slurm job that would run one of the selected applications from the list in table 3.1 with the right amount of iterations to honour the execution time established in the workload. For this transformation an automatic application is used which, depending on some parameters describing the amount of jobs that should be in

Class	Procs	Mem-Bandwith(MB/s)	Application
h	4	4250	synthetichigh
h	8	4250	synthetichigh
h	16	3000	cg.D.16
h	32	3000	cg.D.32
h	64	3000	cg.D.64
m	4	1870	cg.C.4
m	8	1870	cg.C.8
m	16	1870	cg.C.16
m	32	1870	cg.C.32
m	64	1870	cg.C.64
l	2	412	cg.B.2
l	4	412	cg.B.4
l	8	412	cg.B.8
l	16	412	cg.B.16
l	32	412	cg.B.32
l	64	412	cg.B.64

Table 3.1: Applications used in the workload generation

	Medium	High
high	54	84
medium	27	7
low	19	9

Table 3.2: Distribution of applications per workload

each category, randomly selects an application from the application pool and adapts its parameters to the job described in the workload. With this process a series of Slurm jobs is obtained, and for each of them, the arrival time relative to the previous job.

Table 3.2 describes the distribution of jobs according to their classification in both workloads.

Both of the workloads were then run with the three scheduling policies and with different parameters in order to be able to compare the performance of them.

Slurm can allow some grace time to jobs that reach their stated time limit. Without this, all the jobs would be killed once they reach their wall time. This parameter was used to study the impact of the allocation on the execution time of the jobs, an to do so two values for this parameter were used. Firstly, all the workloads were run with the different resource selection policies with a high enough value of this parameter to avoid the killing of any job. This way an analysis can be done on the extra amount of time a job used over the estimation based on the characterization. In second place, all the combinations were run with a value of 5 minutes, a standard value used in some BSC clusters, that allows the jobs to finish if they are near to the end once they reach the time limit.

Chapter 4

Results

As explained before, runs with unlimited time (ie. without killing jobs) were performed in order to analyse the impact of the distribution in the execution time, as well as other runs killing the jobs after 5 minutes of the time limit specified in each job. In both cases, in first place the ConsRes policy and the LessC policy are compared, to see how the specification of the memory bandwidth requirements impacts the usage of the cluster and the execution time of the different applications. Afterwards the LessC and HPerf policies are then compared, followed by a comparison between the HPerf policy starting with an empty database and an HPerf policy with the database preloaded with information from previous executions; and finally the ConsRes and HPerf (preloaded version). This was done for both of the workloads, the High and the Medium.

For the analysis, Paraver traces were generated from Slurm job completion logs in order to visualise the effects of each of the policies and extract some statistics about the different states of the jobs and duration of the executions. On all the applications traces each line represents a job, and each colour represents the job state. The traces show the progressive arrival of jobs into the system and its different states during their life in it. Traces start with the first job arrival and end at the end of the last job running. The red colour represents the time a job is waiting in the queue before starting its execution. The blue colour represents the time a job is running during its required time limit. If the jobs ends before reaching its specified time limit, the resources are freed but this time is marked in white colour to show the extra requested time that was not used. Finally, the dark yellow represents the extra time a job was running beyond the requested time.

4.1 Workload High

4.1.1 Unlimited time

	Duration	Running	Ex. Req	Waiting	Ex. Used
ConsRes	11,460	67,720	1,040	263,648	20,082
LessC	15,424	66,410	2,350	469,604	3,359
HPerf	13,436	66,886	1,874	417,370	7,660
HPerf preloaded	12,846	67,645	1,115	370,182	11,116

Table 4.1: Time in seconds per job state per execution with unlimited time

Figures 4.2a and 4.2b show the trace of the workload execution with ConsRes and LessC respectively. As can be observed in the traces, the waiting time in the first jobs of the LessC execution is higher than on the ConsRes workload. ConsRes has no waiting time until job 19th but in LessC the first job to wait to start is the 7th. Traces also show that on the LessC execution there are more backfilled jobs, ie. job starting sooner than others submitted earlier, which means that LessC has more slots for execution available during the workload. This contrasts with ConsRes, where there are almost no jobs backfilled. Traces also show the considerably amount of extra time used in the case of ConsRes, where almost all the jobs use some extra time beyond their requested time. In the LessC case, there is very few extra time used by jobs and more unused time (white colour) appears in the traces.

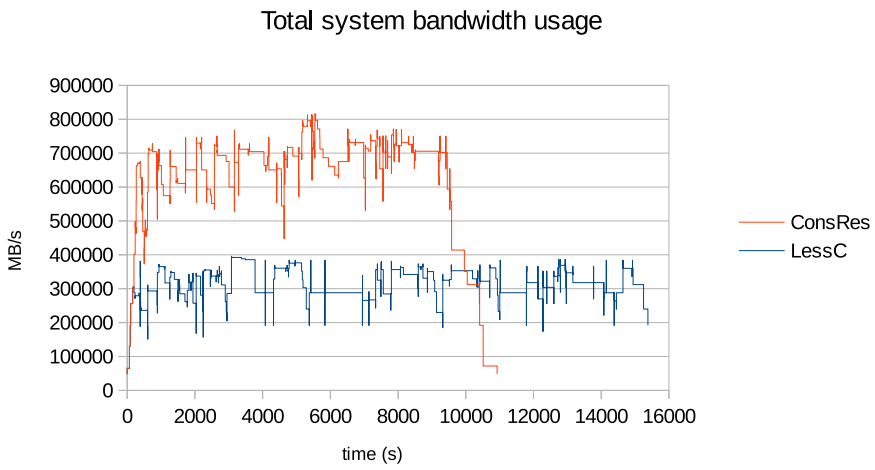
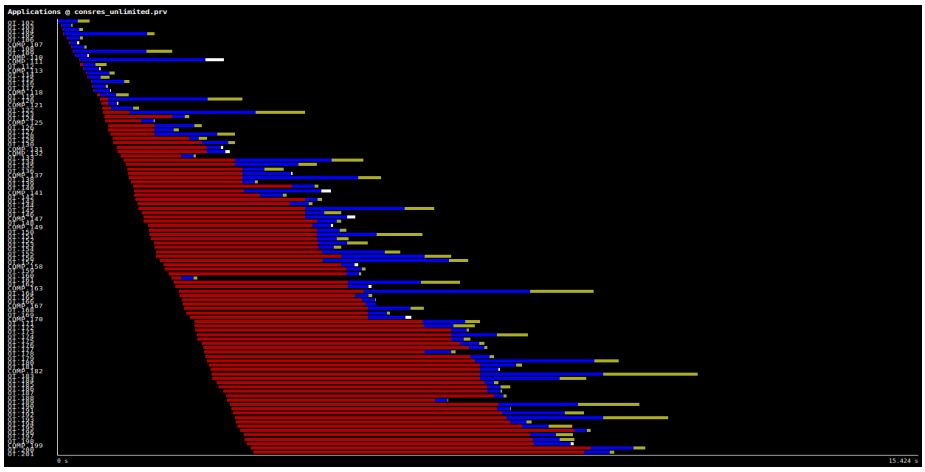
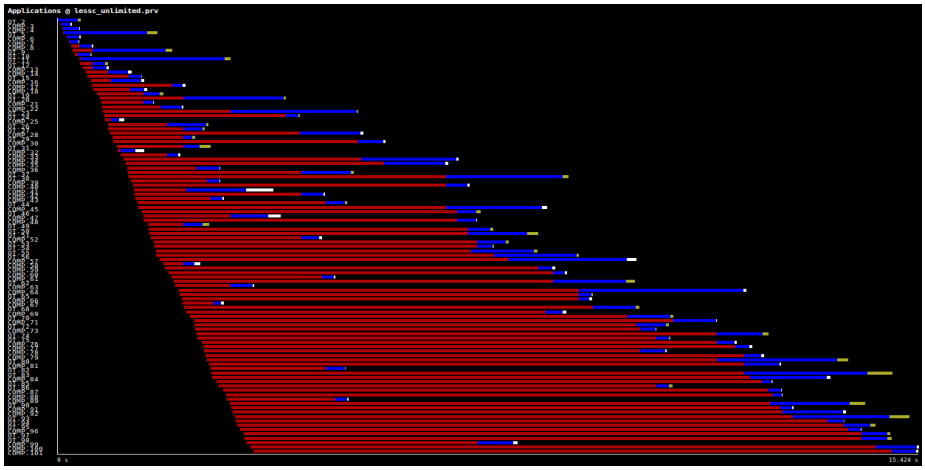


Figure 4.1: Total bandwidth allocated in the system

By not considering the memory bandwidth as a resource, the overall execution time of the ConsRes workload is expected to be shorter than the one with LessC. This behaviour is shown in table 4.1. The total duration of the workload with LessC is a 50% higher than the ConsRes execution. However, the ConsRes extra time used by all the jobs is 6 times the extra time used in LessC. As a



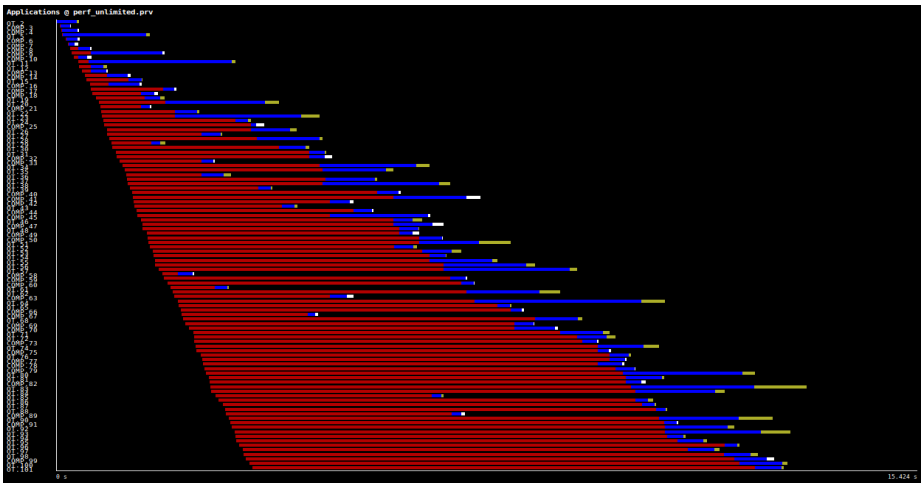
(a) Workload execution with ConsRes and unlimited time



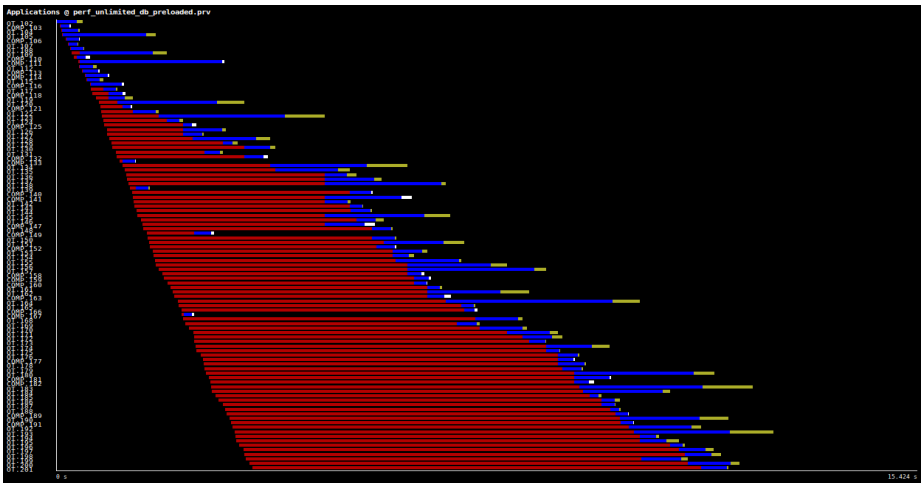
(b) Workload execution with LessC and unlimited time

Figure 4.2

consequence, the requested time by jobs not being used is higher in the LessC execution as jobs do not have to suffer the overload of the nodes as happens in the ConsRes case. The total time spent by jobs running, considering the running time and the extra used time, is largely higher in the ConsRes, having consumed more than 18000 more seconds than LessC. This is due to the large saturation of the compute nodes. This saturation is also the cause of the large difference in total waiting time. Although the total duration of the workload is shorter with ConsRes, the usage of the resources with LessC is more efficient and the applications running time is closer to the expected time limit. The effects of LessC limiting the total amount of memory bandwidth that applications can use on a compute node are shown in figure 4.1. ConsRes with its allocation policy is overloading the nodes by almost the double of the total available memory bandwidth of the cluster, while LessC keeps the maximum allocated memory bandwidth under the total available of 400,000 MB/s.



(a) Workload execution with HPerf and unlimited time



(b) Workload execution with HPerf preloaded and unlimited time

Figure 4.3

When comparing HPerf and LessC both executions look very similar. The duration of the whole execution is slightly shorter in the HPerf policy, although the extra time used by the jobs is higher than in the LessC execution. This difference in the extra time is due to the difficulty to measure in runtime the amount of memory bandwidth being used by an application, which tends to be undermeasured. This also justifies the reduction of the extra requested time as more jobs are finishing after reaching their time limit. However, the total waiting time is reduced significantly.

Comparing now the executions with HPerf and HPerf with a preloaded database, the duration is once again reduced. This is due to the avoidance of exclusive executions for not measured-before applications, which also reduces significantly the overall waiting time. The preloaded workload is the one that highlights the most the effects of the lack of precision in the measurements as shows the

considerably larger amount of extra used time compared to the HPerf.

The traces 4.3a and 4.3b show this described behaviour perfectly. At the beginning of the execution on the HPerf trace, from job 6th jobs start to wait for resources before their execution because of the exclusive execution of unknown applications while in the HPerf preloaded trace there is almost no waiting time until job 16th. This is one of the effects that reduces the overall duration of the workload. Reflected also in the traces is the higher amount of extra used time by the jobs of the preloaded version of the workload and the reduction in the number of jobs that finish their execution before their requested time limit.

Finally, the comparison of ConsRes and Hperf preloaded reveals some interesting information. The total running time of both executions is almost the same although the amount of extra used time is almost the half in the HPerf preloaded case. The extra requested time is again the same while the waiting time is considerably better in the ConsRes case. However, the duration of the HPerf workload is only a 10% higher than that of ConsRes. Figure 4.4 shows the total system bandwidth allocated by ConsRes and HPerf preloaded, and how the HPerf execution is well below the levels of ConsRes.

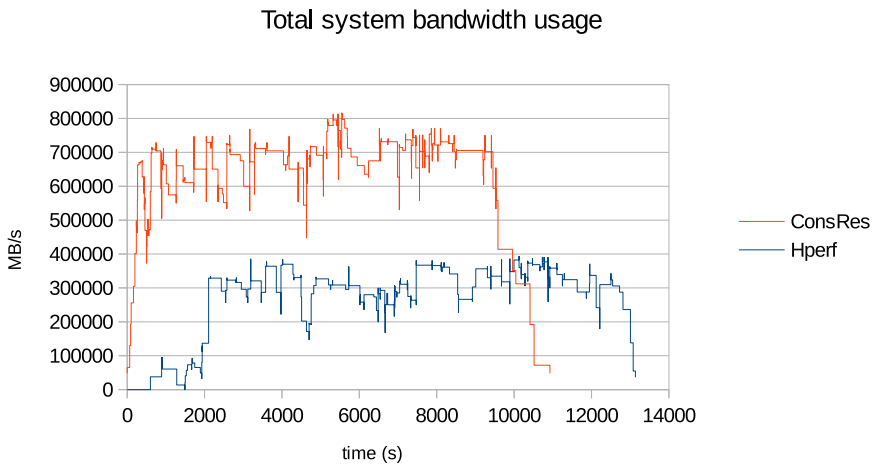


Figure 4.4: Total bandwidth allocated in the system

4.1.2 Limited grace time

	Duration	Running	Ex. Req	Waiting	Ex. Used	Killed
ConsRes	9,914	67,229	1,531	242,795	13,824	26
LessC	15,477	66,707	2,053	466,285	3,265	1
HPerf	14,203	67,768	992	451,478	7,090	9
HPerf preloaded	12,062	67,248	1,512	330,348	8,449	9

Table 4.2: Time in seconds per job state per execution with limited grace time

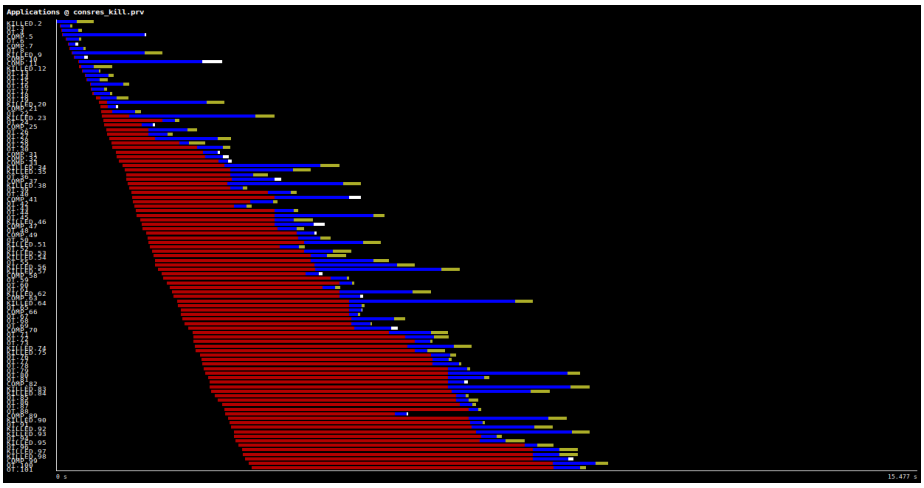
As stated before, runs were performed with a limited grace time, a scenario more similar to production systems, to analyse the impact of the solution proposed in terms of killed jobs.

Comparing ConsRes with LessC, table 4.2 shows again a notably higher duration of the LessC workload in contrast with the duration of the ConsRes case. This is due to the ConsRes lack of control of the memory bandwidth. This leads to an overload of the nodes, as happened in the scenario without limited time, that increases the total running time of the workload, although reducing the total waiting time. It also causes the system to kill far more jobs than the LessC execution. In this case, the difference in running time plus the extra used is not as high as in the previous scenario because of the system killing those jobs that exceed their time limit after 5 minutes. The difference in waiting time continues to be considerable as LessC has almost the same behaviour than before while ConsRes has reduced its total waiting time due to the large number of jobs killed. The job killed in the LessC results from the very tight time limits set in the workload.

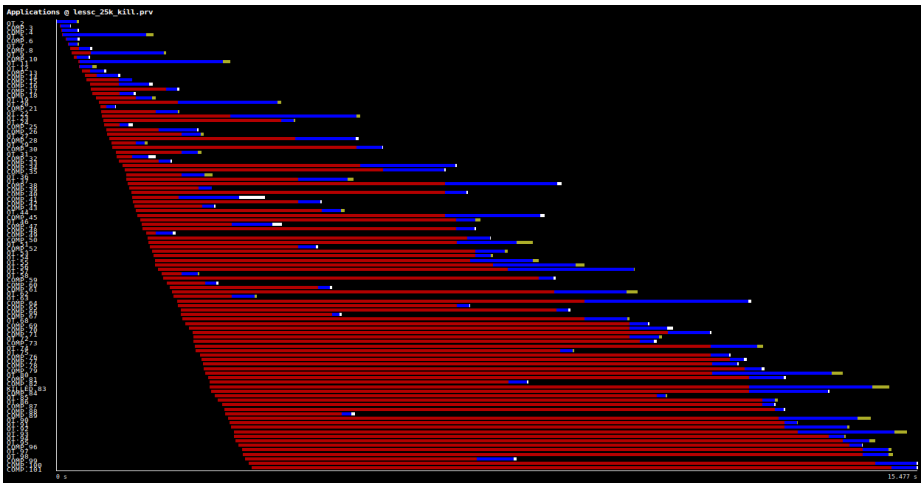
The application traces 4.5a and 4.5b show the behaviour described above. On the ConsRes trace a great number of jobs used all or almost all their extra time resulting in lots of jobs being killed. On the LessC case, however, there is very few extra time used and there are a lot of jobs being backfilled due to the empty slots inherent to this policy.

Changing focus to HPerf and LessC, a very similar behaviour than before is observed. The HPerf has almost the same overall duration than the previous scenario, although little reduced due to the jobs killed. Another consequence of this kills is the reduction in the overall waiting time in comparison with the previous execution. The amount of killed jobs is caused, as explained before, due to the imprecision in the measurements of the memory bandwidth.

The HPerf preloaded execution shows a better duration compared to the non-preloaded execution. Once more the total waiting time of the workload is shorter than those of the HPerf not preloaded and LessC, because of the increase in the jobs killed due to overtime. Comparing it with the ConsRes case the differences are almost the same as in the unlimited scenario. The total duration is higher in the HPerf case, although the extra used time is shorter and the amount of killed jobs is also smaller, resulting in almost the 50% of the jobs killed in the ConsRes experiment.



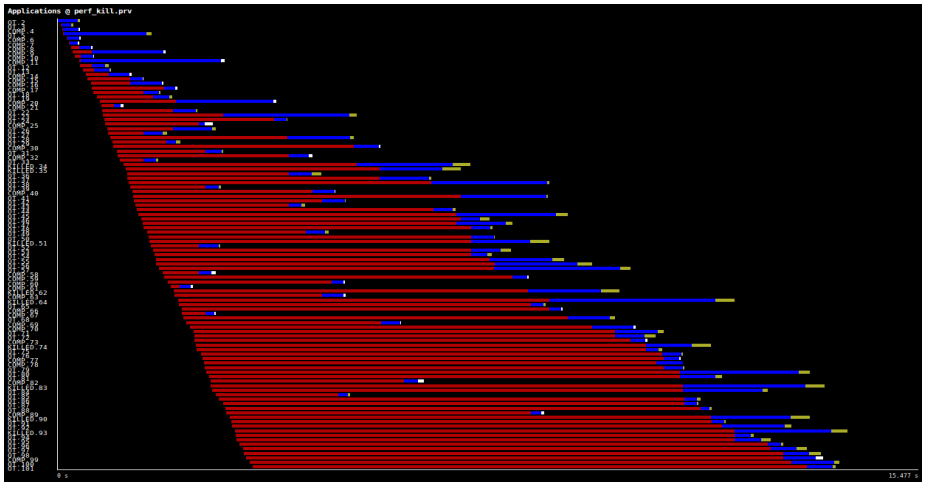
(a) Workload execution with ConsRes and limited grace time



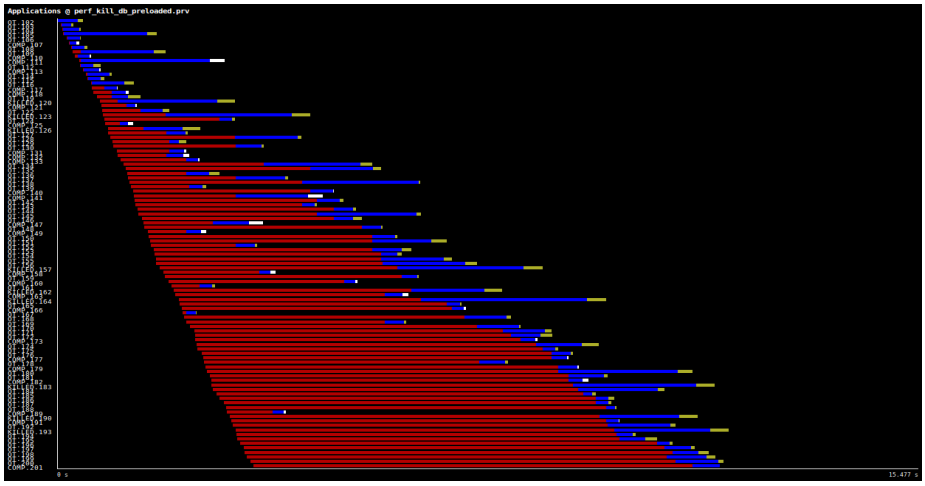
(b) Workload execution with LessC and limited grace time

Figure 4.5

The traces 4.6a and 4.6b show this behaviour. The first jobs of the HPerf execution have more waiting time than those of the preloaded version due to their exclusive execution to get more accurate results. The behaviour at the end of both traces is almost identical, with both executions spending a large amount of extra used time. The backfilling of jobs in both traces is almost the same, leading to the conclusion that the backfilling is not an important factor in the total duration of the trace, but it really is in the total usage of the cluster.



(a) Workload execution with HPerf and limited grace time



(b) Workload execution with HPerf preloaded and limited grace time

Figure 4.6

4.2 Workload Medium

4.2.1 Unlimited time

	Duration	Running	Ex. Req	Waiting	Ex. Used
ConsRes	10,222	66,707	2,113	215,693	13,900
LessC	12,727	64,506	4,314	319,453	5,395
HPerf	12,584	62,996	5,824	354,244	9,931
HPerf preloaded	11,516	67,056	1,764	284,224	14,028

Table 4.3: Time in seconds per job state per execution with unlimited time

The behaviour of the medium workload is very similar to the high one as reflected in table 4.3. ConsRes is the best policy in terms of workload duration. Compared with LessC, the duration is a 20% higher in this last case. However, in terms of running time ConsRes is slightly worse than LessC, as well as in terms of extra used time, where the last is considerably better than ConsRes. The extra requested time is better in the LessC execution than in the ConsRes, which means that the allocation is more efficient in the former case than in the last one. The figure 4.7 shows the bandwidth allocated by both workloads during all the execution. One more time, LessC keeps the bandwidth allocated below the total available, while ConsRes is almost all the time over the threshold.

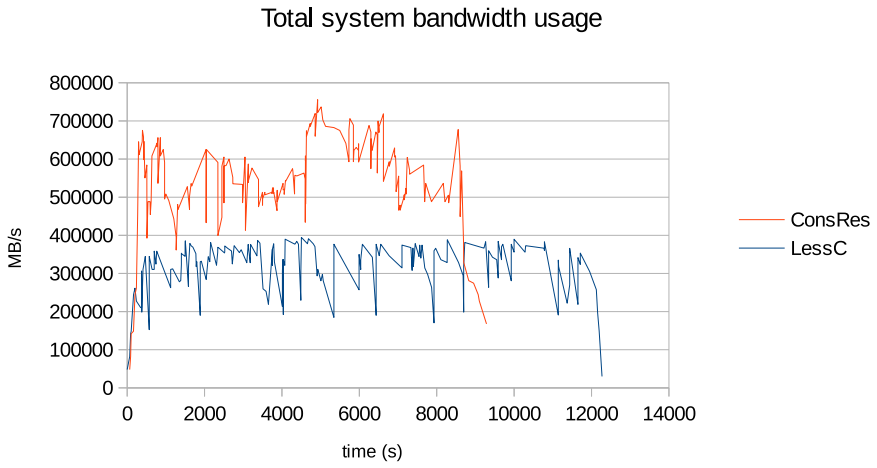
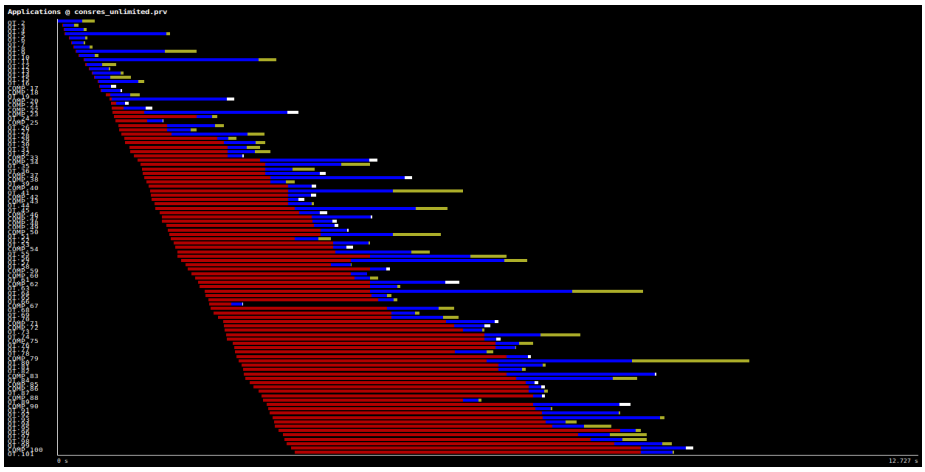
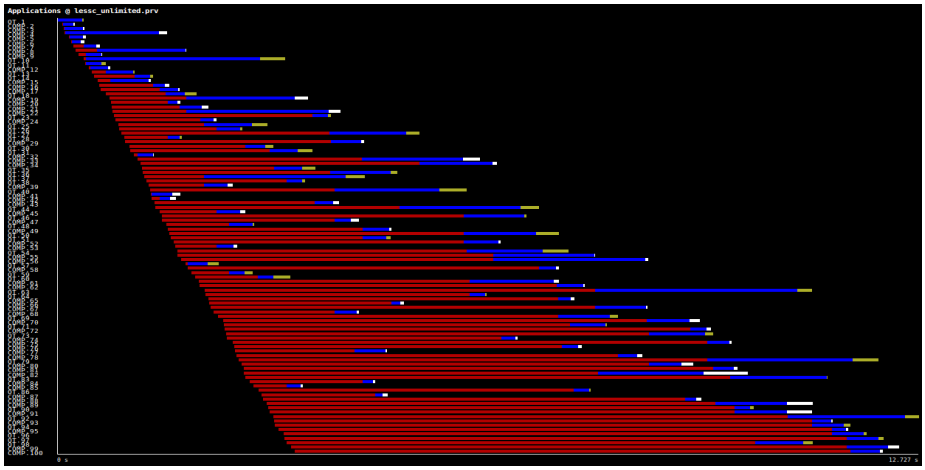


Figure 4.7: MED: Total bandwidth allocated in the system

Traces 4.8a and 4.8b show the behaviour of ConsRes and LessC respectively per job. The large amount of extra time used by the ConsRes jobs and the few jobs that are finishing before their time limit are easily noticed. In the LessC case a great amount of jobs finish before the estimated end time and a lot of jobs are being backfilled to increase the usage of resources. In the ConsRes case



(a) Workload execution with ConsRes and unlimited time



(b) Workload execution with LessC and unlimited time

Figure 4.8

the 17 first jobs do not have to wait for execution while the 7th job in LessC is the first to wait a while for resources. This difference is one of the factors that cause the longer duration of the LessC workload compared to the ConsRes execution.

HPerf has a very similar behaviour to the LessC execution. The duration of both workloads is almost the same. However, it is to highlight the shorter running time of HPerf. This difference in running time is caused by the exclusive executions of the first jobs, which causes some jobs to end sooner due to not sharing any resource with other jobs. However, the exclusive executions also cause the delay in the start of several jobs. This delay, in addition with the inaccuracy of some measures that causes some jobs to run long after their time limit, is what causes both executions to have so similar durations. The delay of the start also increases notably the difference in waiting time from one workload to another.

The difference between the HPerf and the HPerf with a preloaded database is higher than any other case. As expected, the overall duration of the preloaded version is shorter than the HPerf execution. The difference in running time is also the highest between all the executions, as well as the difference in the extra requested time. This is caused by some imprecise measurements, as commented before, that cause some of the jobs to last more than expected. The difference in the waiting time between both executions is caused again by the exclusive executions of the HPerf workload in contrast with the preloaded version that has no exclusive job.

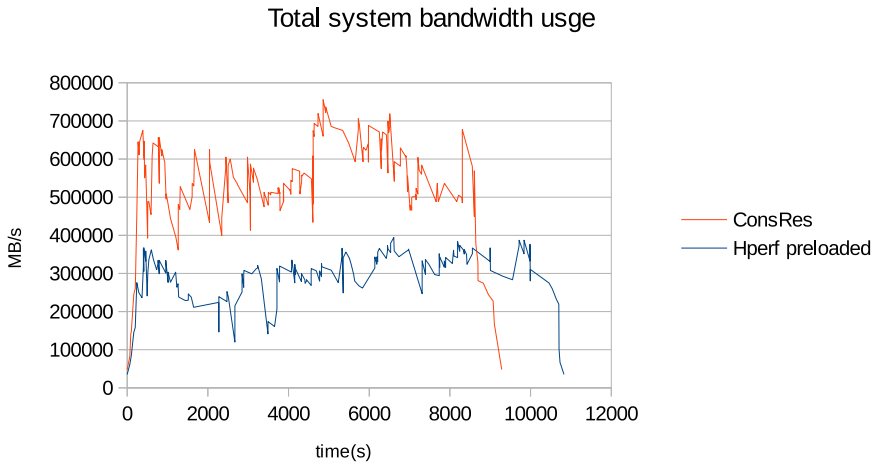
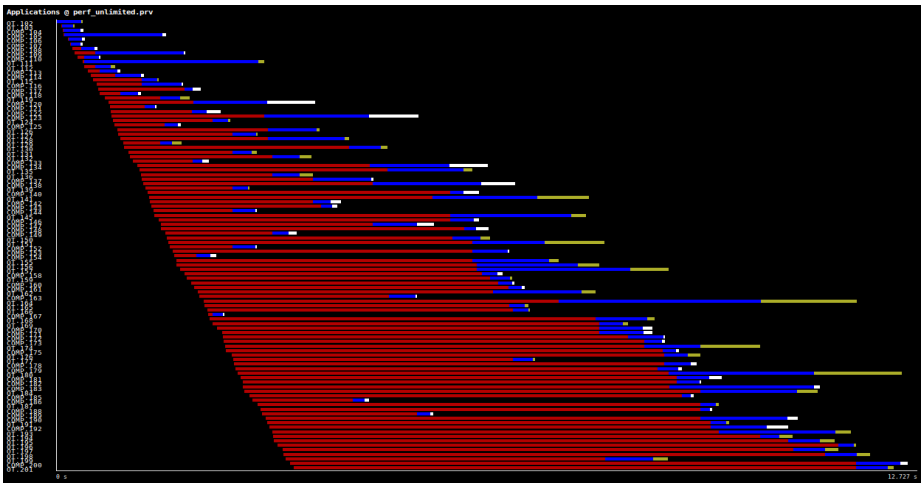


Figure 4.9: MED: Total bandwidth allocated in the system

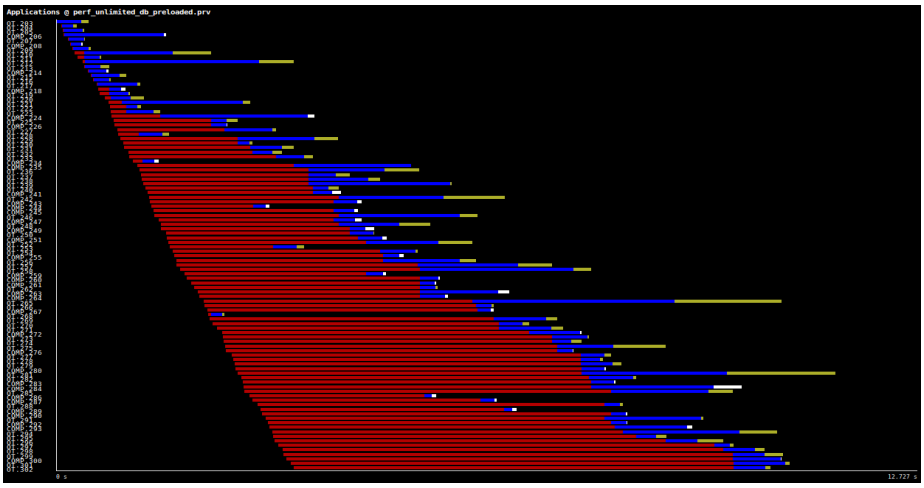
Traces 4.10a and 4.10b reflect the comments above. The HPerf execution shows that jobs start to wait very early while the jobs of the preloaded workload do not start to wait until job 15th. The amount of extra requested time is very clear in the HPerf trace, and is almost inexistent in the preloaded execution. In both traces appear the effects of the poor accuracy of the measuring system that causes the large extra used time in the last jobs of each trace.

Finally, comparing the ConsRes execution with the HPerf preloaded version the differences are almost inexistent. The duration of the whole execution is slightly higher on the HPerf execution, as well as the running time and the extra used time, for the reasons already commented. The waiting time is also notably higher, due to ConsRes overloading the nodes in terms of memory bandwidth.

Figure 4.9 shows the comparison between the bandwidth allocated by ConsRes and HPerf during the whole execution of the workload. Once again ConsRes does not consider the memory bandwidth as a resource and overloads the system, while the HPerf execution is always below the system limit.



(a) Workload execution with HPerf and unlimited time



(b) Workload execution with HPerf preloaded and unlimited time

Figure 4.10

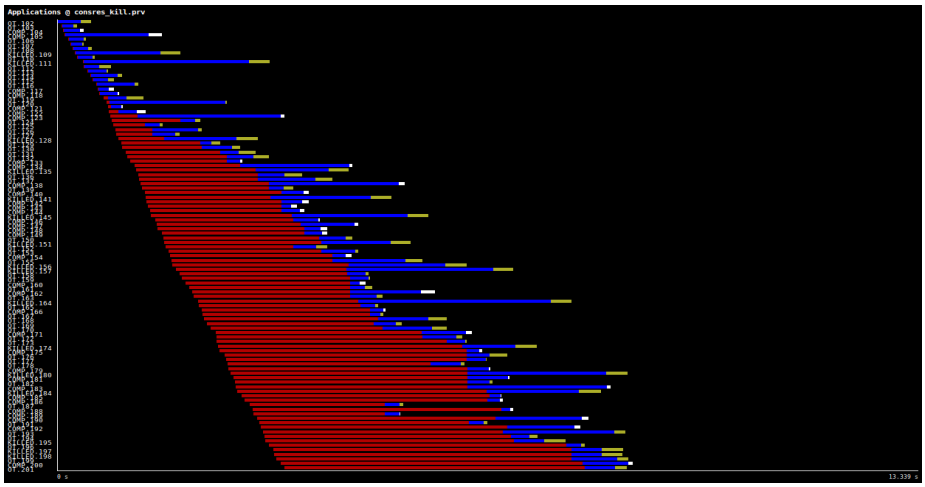
4.2.2 Limited grace time

	Duration	Running	Ex. Req	Waiting	Ex. Used	Killed
ConsRes	8,905	66,778	2,042	211,052	10,067	16
LessC	13,339	64,557	4,263	311,197	4,721	6
HPerf	11,518	64,405	4,415	354,626	6,410	8
HPerf preloaded	9,318	66,384	2,436	220,800	9,683	13

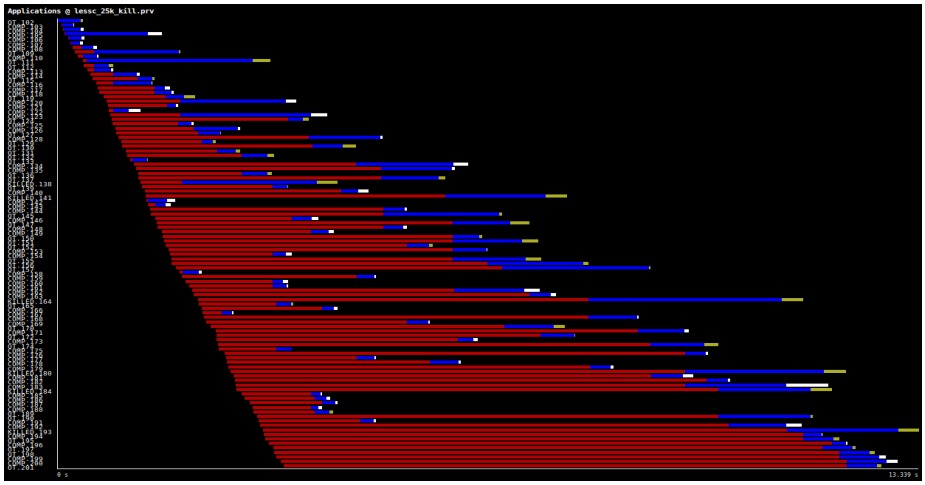
Table 4.4: Time in seconds per job state per execution with limited grace time

The executions with a limited extra time per job are resumed in table 4.4. ConsRes execution reduces considerably the total duration of the workload due to the high number of jobs killed for exceeding the job time limit and the grace

time. Another consequence of killing jobs is the reduction of the total waiting time. For the LessC case, the total workload duration is slightly increased although the rest of the metrics keep very similar values to those in the previous scenario. In this case, although having killed some jobs the duration does not decrease but increases. This is caused by different allocations in both executions although the differences are not significative. Comparing traces 4.8b and 4.11b there is no clear reason for the increase of the duration in the limited time case, and may be due to system noise.



(a) Workload execution with ConsRes and limited grace time



(b) Workload execution with LessC and limited grace time

Figure 4.11

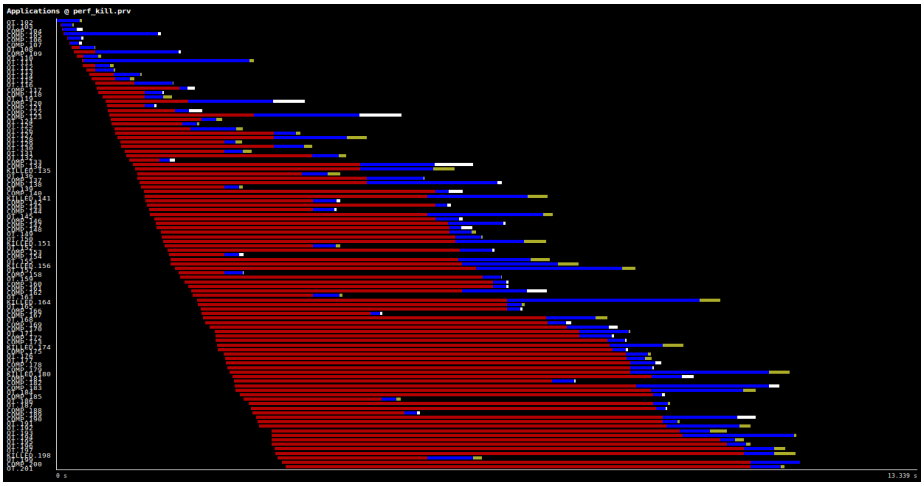
Comparing the traces of ConsRes and LessC, figures 4.11a and 4.11b respectively, there is a high number of jobs using extra time in the ConsRes execution while this usage is very limited in the LessC case. Also the backfill effects are notable on the LessC case, as there are lots of jobs starting before their expected starting time according to their position in the queue.

Comparing now LessC and HPerf, the later performs better in this run. The total workload duration is reduced, from similar values in the unlimited time runs to almost 2000 seconds difference in this one. The total waiting time, however, is higher in the HPerf case than in the LessC, but is around the same values as in the previous execution. The number of killed jobs is slightly higher, being this the main reason for the reduction in the overall duration.

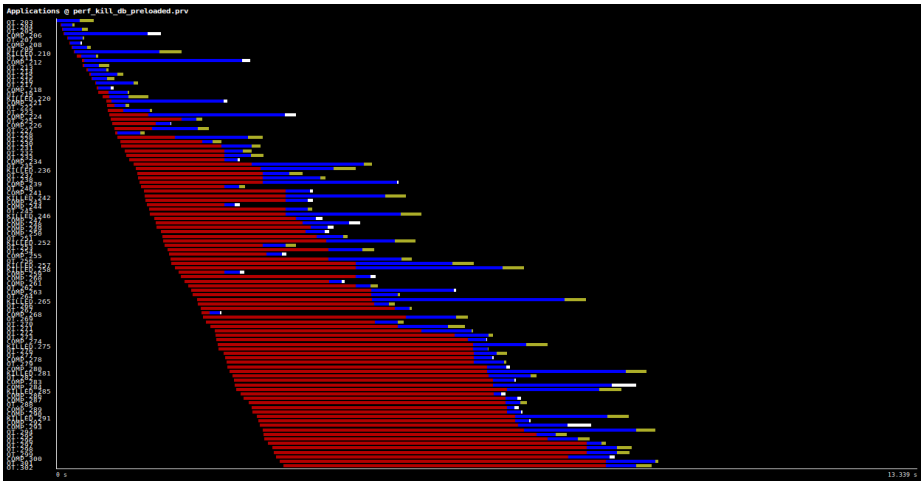
For the HPerf and HPerf with preloaded database cases, the numbers are relatively better in the preloaded case. The workload duration is shorter in the preloaded case than in the HPerf case, and the waiting time is reduced considerably because of the exclusive executions of the first jobs in the HPerf case. As a consequence, the extra requested time in the HPerf case is higher than with the full database, and the extra used time is lower. Also the amount of killed jobs is lower in the HPerf case than in the preloaded case.

Traces 4.12a and 4.12b show the HPerf and HPerf preloaded respectively. They show once more the waiting time in the first jobs of the HPerf workload, while there is almost no waiting time in the preloaded workload. The extra requested time is clearly visible in several jobs of the HPerf execution, but almost non existent in the full database case. For the extra used time the opposite is shown: the HPerf uses very little amount of extra time while in the HPerf preloaded workload almost all the jobs spent some time after their requested time limit.

The HPerf preloaded execution is very close in numbers to the ConsRes workload. The duration is slightly higher, as well as the waiting time, but the number of killed jobs is lower in the HPerf case than in the ConsRes execution.



(a) Workload execution with HPerf and limited grace time



(b) Workload execution with HPerf preloaded and limited grace time

Figure 4.12

Chapter 5

Conclusions and future work

This work shows how the use of a monitoring system can help a job scheduler to improve the overall scheduling of jobs to reduce the waste of resources and the number of killed jobs due to reaching their time limit. The proposed solution uses a transparent monitoring system to obtain the performance information from the applications without adding a considerable overhead to its execution time. The use of the data collected helps the scheduler to better allocate jobs to target resources and achieve a reduced execution time. The existing solution of previous studies adds too much duration to the overall workload to be a feasible solution. It also requires the users to specify the amount of required resources although they may not know exactly how much use of the shared resources their applications need. Our solution solves this by asking the users for a tag to identify the job, instead of having to specify a quantitative amount of resources. This mechanism prevents the knowledge problem from users point of view, and avoids the fraudulent statements from users to achieve a sooner execution.

The solution proposed has a better performance in terms of workload duration than the more informed policy (by users), and slightly higher duration than the default policy that does not consider at all the resource analysed in this study. Despite being slower than the default scheduling policy, the amount of jobs killed due to time out is reduced significantly in both of the scenarios studied. The higher the amount of memory bandwidth the job uses, the better the performance of the solution.

Despite not being able to accurately measure the amount of memory bandwidth used by the applications, which should be the topic for future research, the solution proposed in this work is proven to be a feasible solution for a production machine, and with a more accurate mechanism to obtain the information, the results would be even better than the ones shown in the present work.

References

[1] Xavier Abellán Écija, Jesus Labarta Mancho, and David Vicente Dorca. *Adaptació i desenvolupament d'un sistema de mesures de rendiment per a MareNostrum.* 2009.

[2] Carlos Fenoy García and Julita Corbalán González. *Millora de la gestió de recursos a SLURM.* 2010.

[3] Francesc Guim Bernat and Universitat Politècnica de Catalunya. Departament d'Arquitectura de Computadors. *Job-guided Scheduling Strategies for Multi-site HPC Infrastructures.* 2008.

[4] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[5] PhilipJ. Mucci, Daniel Ahlin, Johan Danielsson, Per Ekman, and Lars Malinowski. Perfminer: Cluster-wide collection, storage and presentation of application level hardware performance data. In JoséC. Cunha and PedroD. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 124–133. Springer Berlin Heidelberg, 2005.

[6] Rob VanderWijngaart and Bryan A Biegel. Nas parallel benchmarks. 2.4. 2002.