DJSB: Dynamic Job Scheduling Benchmark

Victor Lopez[1], Ana Jokanovic[1], Marco D'Amico[1], Marta Garcia[1], Raul Sirvent[1], and Julita Corbalan[2]

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Universitat Politecnica de Catalunya, Barcelona, Spain

Abstract. High-performance computing (HPC) systems are very big and powerful systems, with the main goal of achieving maximum performance of parallel jobs. Many dynamic factors influence the performance which makes this goal a non-trivial task. According to our knowledge, there is no standard tool to automatize performance evaluation through comparing different configurations and helping system administrators to select the best scheduling policy or the best job scheduler. This paper presents the Dynamic Job Scheduler Benchmark (DJSB). It is a configurable tool that compares performance metrics for different scenarios. DJSB receives a workload description and some general arguments such as job submission commands and generates performance metrics and performance plots. To test and present DJSB, we have compared three different scenarios with dynamic resource management strategies using DJSB experiment-driven tool. Results show that just changing some DJSB arguments we can set up and execute quite different experiments, making easy the comparison. In this particular case, a cooperative-dynamic resource management is evaluated compared with other resource management approaches.

Keywords: dynamic resource management, job scheduling, benchmark, performance evaluation

1 Introduction and Motivation

HPC systems are big systems with very powerful computational and communication capacities, specially designed for parallel applications with high requirements in terms of computation and inter-process communication. This specific hardware makes HPC systems very expensive and complex, resulting in the necessity of expert software systems, i.e., *job schedulers* to deal with the job scheduling and resource allocation. Additionally, system administrators configure and control system behaviour. An example of job schedulers used in the top five HPC systems are SLURM [20], PBS [14], or Cobalt [9].

The complexity of HPC systems has grown with their size, as well as with the jobs complexity. They are composed of nodes with many cores and GPUs. The resources are shared among jobs at different levels: memory, network, etc. being hierarchically organized. As a consequence, parallel jobs have also evolved to hybrid programming models to fit this configuration. Most of the jobs executed

The final publication is available at Springer via https://link.springer.com/chapter/10.1007/978-3-319-77398-8_10

in these systems are programmed in pure MPI [4], OpenMP [8] or OmpSs [6], or hybrid models such as MPI+OmpSS.

Job schedulers allow system administrators to configure the machine with different partitions, policies, policy arguments, etc. Users can also configure their job submissions with as many requirements and details as needed for a *"perfect"* job execution. Job schedulers try to execute jobs as soon as possible based on the job requirements, priorities (e.g., arrival order), and resource availability. If resource requirements are very specific to improve execution time, that may increase wait time, resulting in a poor global performance, i.e., slowdown. If job requirements are flexible, jobs can start before but their execution time can suffer variations because of sharing of resources such as network bandwidth, for example.

Traditional approach in HPC systems is to statically allocate resources to jobs once they are started, i.e., they are not preempted and they own these resources until the end of their execution. This approach simplifies job management but reduces potential performance improvements that can be achieved with dynamic approaches. The deployment and evaluation of dynamic job scheduling strategies is complicated and it is a normal approach for system administrators when upgrading their systems, or starting new HPC centers, to select well known static approaches rather than evaluating different dynamic strategies and selecting the one with best performance. This evaluation must be based on center characteristics and specific workload.

The aim of this paper is to present the Dynamic Job Scheduler Benchmark (DJSB). DJSB is a tool that evaluates the capacity of system to react to the resource requirements of the jobs that may overload the system, which we will refer to as the *dynamicity* of the system.

DJSB can be configured to deal with different job schedulers, as well as, interactive sessions that do not use job schedulers, different job submission frequencies and different application arguments such as number of tasks. Early prototype of the benchmark has already proven its usefulness and has been used by other research groups [7].

DJSB actual workload is based on a use case defined as a reference case in the Human Brain Project [2]. The use case consists in a situation where there is a big and long running job using all the resources and a new, small and short job that arrives to the system requesting a percentage of these resources during a short period of time. In this use case, the second job does a partial analysis of results reported by the simulation. Therefore, we will refer to the long running job as the simulation and to the new job as the analytics.

To illustrate the potential of DJSB, we have performed three different sets of experiments, each one including many variations concerning system size, application size, number of applications, memory requirements, etc. We have compared a stop&continue approach with oversubscription and a cooperative-dynamic resource management.

To evaluate the benefits of such a dynamic environment we will present traditional performance metric slowdown but also a new synthesized metric that

combines slowdown of both jobs and tries to summarize in a single value the *dynamicity* of a system. This metric is presented in Sections 2 and 4.

The rest of the paper is organized as follows: Section 2 describes DJSB tool, experimental setup and the metrics reported. Section 3 describes our dynamic resource management execution environment. Section 4 presents evaluation results comparing among the different scenarios. Section 5 presents related work, from the point of view of benchmarks and dynamic scheduling evaluation. Finally, section 6 presents conclusions and future work.

2 DJSB: Dynamic Job Scheduler Benchmark

The purpose of the dynamic scheduling benchmark is to do an automatic performance comparison of different solutions on pilot systems. It provides a synthetic model of a hypothetic interactive session workload. DJSB is implemented as a Python (configurable) script that drives application execution, monitoring, metric collection and generation of performance metrics and graphs.

Some mock-up parallel applications are provided to represent two types of applications: a single long running simulation and a potential in situ analysis. The benchmark will measure the impact of one application on the other and the decrease of their performances. The focus will be on the ability to support dynamic scheduling policies, so I/O will be minimal.

DJSB behaviour is configured based on several configurable parts:

- **General options.** This component defines the global DJSB experiment. It includes arguments such as the number of samples of applications.
- **Job submission options**. DJSB can be executed in systems with different queueing systems. The basic job submission and monitoring commands can be specified.
- **Application options**. Specific details for the long running simulation and the analytics can be specified such as execution time or memory consumption.

DJSB assumes applications are previously compiled. Once it starts, it computes reference metrics, that is, execution time of each application when running alone based on Application options. This is the *reference stage*. Once references are available, it starts the job submission based on Job submission options and General options. This *execution stage* includes the re-execution of the experiment several times to provide statistically significant measurements. Once execution stage finishes, a performance metrics file is generated together with the plots to make easy performance evaluation.

2.1 General options

Some of the most relevant general options are:

- **num_of_samples** Number of samples or repetitions. One repetition implies one execution of the whole benchmark (simulation + analysis).

– **sleep_min_time,sleep_max_time** Minimum/Maximum sleep time between each analysis in the execution stage sample.
– **num_of_ref_analysis** Number of total analytics to be executed per sample of the reference stage.
– **num_of_dyn_analysis** Number of total analytics to be executed per sample of the execution stage.

2.2 Job submission options

DJSB uses a job (Python) module where an API to deal with job submission and monitoring is specified. This API supports jobs executed in an interactive session, or submitted in a previously created reservation, together with the interaction with job schedulers such as SLURM or LSF[21]. Commands to be implemented in the job module are the next one (a job module template is provided):

– **get_submit_command(self)**: The method must return the shell command to submit a job. The command can be complemented by the class attributes from the constructor, such as the total number of tasks, etc.
– **get_poll_completion_command(self, submit_stdout, submit_stderr)**: The benchmark needs to poll the system until the job completion occurs. This command returns a shell command to test the condition.
– **get_{suspend,resume}_command**: To be executed when suspending or resuming a job

2.3 Application options

DJSB is developed to support different applications but the results presented in this paper are based on different configurations of the STREAM benchmark since it was a requirement of the Human Brain Project [2] in its previous stage.

– **A single, long running, simulation job.** The preferred mock-up application will be based on a version of the STREAM benchmark written in Fortran/C [16]. Parallel versions using MPI only and MPI plus OpenMP (based on parallel loop) are provided. The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels.
– **Application analytics:** For analytics we use the same approach and application. We use a small version of STREAM executed periodically in the middle of the big STREAM and requesting part of the resources used by the big one.

Arguments to provide detailed application descriptions are:

– **total_tasks** Number of tasks (MPI processes) for the specific application.
– **cpus_per_task** Number of CPUs per task for the application. Also, number of OpenMP threads per MPI process.

- **same_nodes_as_sim** Only valid for the analysis application. If true, the analysis will be submitted in the same nodes as the simulation, as long as the job interface allows it.
- **total_memory** Total amount of memory to be used for the application
- **exec_time** Estimated execution time of each instance of the application
- **node_host_names** List of nodes where the application is allowed to run. The list is forwarded to the job module. If the list is empty, the job scheduler should decide the allocation (default option).
- **command** Path to the application binary.

2.4 Application performance metrics

During reference stage, DJSB collects reference execution time per application, both simulation and analytics. This reference time is computed as the average of the several executions performed at the reference stage. During the execution stage, traditional scheduling metrics are computed such as wait time, response time, or slowdown.

1. Wait time - the time elapsed between the job submission and the start of the job.
2. Execution time - the time elapsed between the start of the application and its completion.
3. Response time - the time elapsed between the job submission and the job completion (wait time + execution time).
4. Slowdown - the ratio between the response time when the application is executed in a workload, i.e., sharing resources with other applications, $T^{sharing}$, and the execution time of the application executed alone on the exclusive resources, T^{alone}_{REF}, that is collected in reference stage.

$$slowdown = \frac{T^{sharing}}{T^{alone}_{REF}} \tag{1}$$

Along with these metrics, DJSB computes *application weights*, which are equal to the resources the application was occupying during its execution multiplied by the execution time of the application. The benchmark calculates the weights for each application and for each scenario. The formulae for calculating the weights for each specific scenario are given in the Section 4 along with the description of the scenarios.

2.5 Workload metrics

Based on individual application metrics, traditional workload metrics are provided such as average wait time, average slowdown and average response time, along with specific DJSB metric such as the *dynamicity*. The dynamicity will greatly depend on the system capability to manage and run different jobs at the same time. For calculating dynamicity, DJSB uses weighted geometric mean,

suggested in various works [13], [17], [15] to be used for comparing among the systems when using relative values such as slowdown. The more capable the system is to accommodate the applications without high performance penalty, the dynamicity should be higher. Therefore, we use inverse weighted geometric mean of the applications slowdowns. For a use case workload, consisting of a simulation and an analytics, calculating the dynamicity of a specific system, i.e., scenario reduces to calculating the formula 2. The weights w_s and w_a are for simulation and analytics, respectively, and are calculated for each scenario differently. The $slowdown_s$ and $slowdown_a$ are the slowdowns of simulation and analytics, respectively. Thus, we will get a dynamicity value for each scenario, which allows us to compare different scenarios.

$$dynamicity = e^{-\frac{w_s \cdot \ln slowdown_s + w_a \cdot \ln slowdown_a}{w_s + w_a}} \quad (2)$$

3 Cooperative dynamic resource management

Dynamic scheduling has been a research topic for many years. In this section we describe the execution environment used in this work, as well as, dynamic resource management scenario that we will use in the experiments.

Figure 1 shows the main components of our execution environment. The main characteristic of our execution environment is that the job scheduler and the resource manager, in this case SLURM, cooperate with an additional runtime library, Dynamic Load Balancer (DLB), that helps the system to exploit malleability in an efficient way. One of the main components of DLB is Dynamic Resource Ownership Manager (DROM).

The execution environment is composed by the following software components :

- **Job scheduler i.e., SLURM controller.** SLURM is composed of two components, the SLURM controller and a SLURM daemon per node. The job scheduler is implemented by SLURM controller, it is in charge of job submissions. It receives job requirements and it decides *when* and *where* a job can be started based on its requirements, scheduling policy and system status.
- **Node manager, i.e., SLURM daemon extended with DLB-DROM component.** Each Node manager is aware of the number of jobs and processes being executed in the node. The Node manager provides resource management services offered by SLURM extended with DLB-DROM API for process ownership management. DLB ownership mechanism gives a possibility for a flexible resource allocation, where processing cores can be used by processes that do not own them during the cores' idle periods.
- **Programming model libraries, i.e., MPI/OpenMP/OmpSs.** These three programming models are transparently supported. Malleability is easily supported in OpenMP [8] and OmpSs [11]. Malleability has been also proposed for MPI in different contexts: Virtual malleability was proposed for MPI in [18], [12] and it is also included MPI-3. However, even having

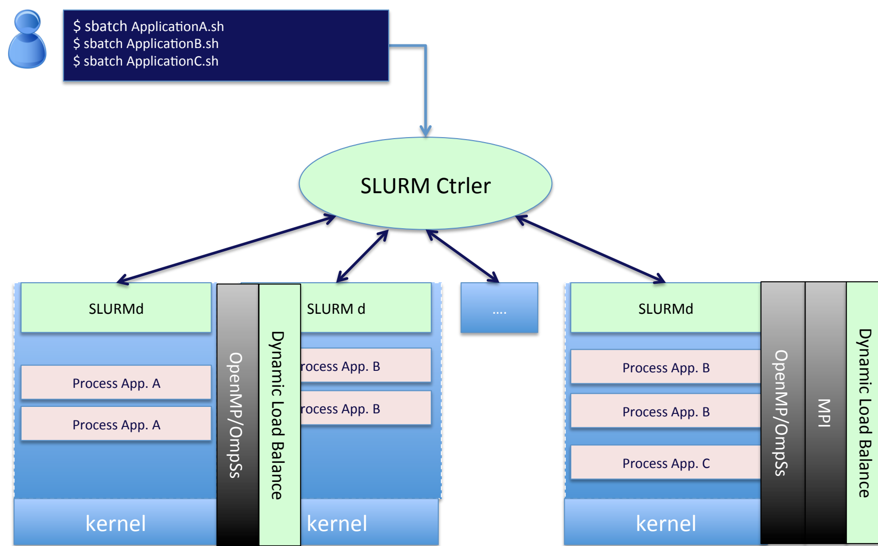


Fig. 1: Cooperative Dynamic Resource Management

these proposals, it is a normal practice to exploit the malleability by using a second level of parallelism and using OpenMP in it. DLB-DROM supports MPI+OpeMP/OmpSs or only OmpSs as indicated in Figure 1.

Cooperative-dynamic resource management scenario in case of our simulation-analytics workload works as follows. When the analytics arrives to the system, the simulation resources are shrunk to accommodate the analytics on as much resources as it requests. The simulation resources are practically lent to analytics for some period of time. This dynamic redistribution of resources among simulation and analytics is done by our SLURM-DROM environment. As soon as analytics finishes its execution, the resources are returned to simulation, and the simulation is expanded, using all of its resources. The same scenario repeats when the new analytics arrives to the system. Section 4 presents the explained scenario along with the other evaluated scenarios and gives the graphical view in the Figure 2.

4 Evaluation

4.1 Scenarios

Three different scenarios are used to evaluate the *dynamicity* of the system. Dinamicity is defined as the capacity to react to workload changes and reallocate resources to running jobs in order to minimize expected slowdown. Figure 2 shows these three scenarios.

- **Oversubscription.** This is a scenario where the simulation has been previously started by the scheduler and the analytics jobs are submitted to the same job reservation, thus positively reducing the wait time to zero, but sharing resources with the simulation. The sharing of resources is fully controlled by operating system. This is untypical scenario in HPC environment and it is enabled by configuring SLURM to force resource sharing. Typical scenario would be the one where each job waits in the queue until enough resources are available. Since analytics jobs need to be executed along with simulation, the typical scenario is not applicable in this use case.
- **Stop&Continue.** When the analytics job is submitted, the already running simulation job is stopped. The analytics job starts without waiting for resources. The old job remains in memory. This is not a problem in case the memory is not a critical resource. The overhead in this scenario comes from the time required to stop/resume processes and from the memory that may be overloaded. We have used SLURM's [20] suspend/resume mechanism for this scenario.
- **Cooperative-Dynamic Resource Management.** Scenario described in the previous section.

4.2 Configurations

Configuration parameters of DJSB benchmark are given in Table 1. Configuration parameters for simulation and analytics are given in Table 2.

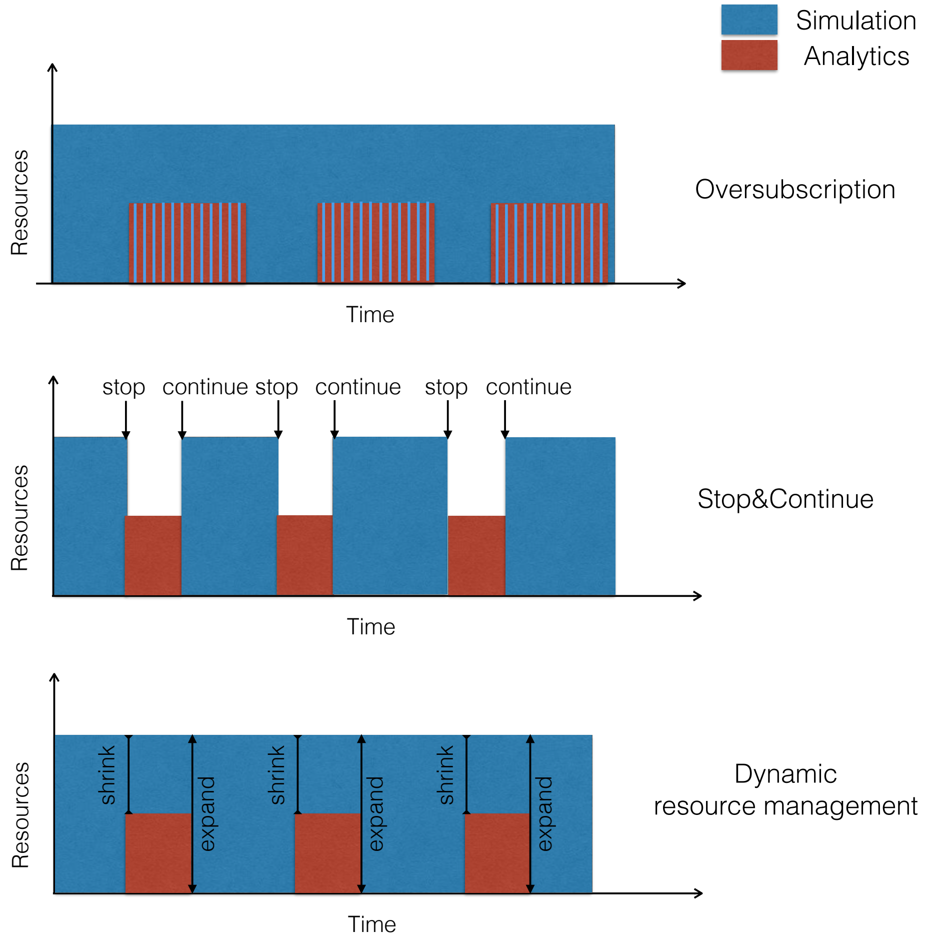


Fig. 2: Three scenarios: oversubscription, stop&continue, and dynamic resource management. A single execution of simulation is performed, while multiple instances of analytics are submitted to the same resources over time of simulation's execution.

Argument or Module	Value
Number of samples	3
Number of simulation runs per sample	1
Number of analytics runs per sample (reference stage)	2
Number of analytics runs per sample (execution stage)	1, 2, 4, and 6

Table 1: DJSB configuration parameters

Argument	Simulation	Analytics
Duration	5 min	10 s
Job size	512 or 1024 CPUs	50% of simulation size
Tasks per node	4	2
Memory per task	1GiB, 4GiB and 5 GiB	50 MiB
OpenMP threads	4 per task	4 per task

Table 2: Simulation and analytics configuration parameters

4.3 Metrics

For evaluating the results we will use slowdown and dynamicity, already explained in the Section 2. Here we give the formulae for calculating weights for simulation and analytics for each of the scenarios. We use the following notation:

- N^s_{CPUS} - Number of CPUs used by simulation
- N^a_{CPUS} - Number of CPUs used by analytics
- $T^{scenario}_s$ - Execution time of simulation for a given scenario
- $T^{scenario}_a$ - Execution time of analytics for a given scenario
- n_a - Number of analytics runs during a single simulation execution

In case of Oversubscription scenario DJSB uses the following formulae:

$$w^{oversubs}_s = N^s_{CPUS} \cdot T^{oversubs}_s - \frac{1}{2} \cdot N^a_{CPUS} \cdot n_a \cdot T^{oversubs}_a \tag{3}$$

$$w^{oversubs}_a = \frac{1}{2} \cdot N^a_{CPUS} \cdot T^{oversubs}_a \tag{4}$$

In case of Stop&Continue scenario DJSB uses the following formulae:

$$w^{stopcont}_s = N^s_{CPUS} \cdot (T^{stopcont}_s - n_a \cdot T^{stopcont}_a) \tag{5}$$

$$w^{stopcont}_a = N^a_{CPUS} \cdot T^{stopcont}_a \tag{6}$$

In case of Dynamic scenario DJSB uses the following formulae:

$$w^{dynamic}_s = N^s_{CPUS} \cdot T^{dynamic}_s - N^a_{CPUS} \cdot n_a \cdot T^{dynamic}_a \tag{7}$$

$$w^{dynamic}_a = N^a_{CPUS} \cdot T^{dynamic}_a \tag{8}$$

4.4 Results

In order to show the usefulness of DJSB, we have performed a set of real experiments on the local, MareNostrum supercomputer [3]. MareNostrum consists of 3098 computing nodes, with 16 processing cores per node. For our experiments we requested the computing nodes with 32GB of main memory per node.

We have evaluated the impact of the following parameters on the performance of the simulation, the analytics and the system:

- The number of the analytics jobs
- Memory per task of simulation
- The size of the system

Figure 3 (a) shows that slowdown of simulation increases as the number of analytics that it shares resources with increases. The least impact is in the case of dynamic scenario – at most 20% of performance loss. The highest impact on simulation's performance is in the case of oversubscription scenario, up to 20% worse than in the case of stop&continue, and up to 30% worse than in case of dynamic scenario. As Figure 3 (b) shows, the analytics job is the most impacted in the case of oversubscription up to 214%. It is the least impacted in the case of stop&continue scenario, as the simulation is stopped during its execution and the total request for memory per node by both applications is less than 15% of the total node memory. Dynamic scenario leads to up to 104% loss of analytics performance. We present analytics results for each of the experiments with different number of analytics (x-axis), but as expected, the analytics does not change performance depending on how much instances of analytics have been run. We present average of all the analytics run within a single experiment. Regarding the system dinamicity, Figure 3 (c) shows that as number of analytics increases in the workload in general the system is less capable to mange the load in an effective way. In particular, dynamicity in case of oversubscription goes as low as 62%, the highest is in case of stop&continue – at least 79%, and in the case of dynamic, the dynamicity is at least 74%.

Further we configured the benchmark, i.e., the simulation options to request more memory. The total memory requested by simulation and analytics per node was more than 50% of total node memory. Figure 4 shows the same set of experiments in the case of higher memory demand. Regarding the slowdowns of the application, the simulation at most 20% more impacted when its demand for memory is higher, whereas, analytics suffers significant impact of almost a double performance loss comparing to the previous set of experiments. The dynamicity plot shows that the impact of increase in memory requirements by simulation makes system less capable to deal with the new coming applications. The dynamicity goes as low as 52%, 63% and 49% , for oversubscription, stop&continue and dynamic scenario, respectively. While dynamic scenario might be good for the system with computation-intesive applications, in case of dominantly memory-intesive applications, such as STREAM, the dynamic resource management does not bring better performance than oversubscription scenario.

Finally, we configured benchmark to test the smaller system size, i.e., 512 CPUS. Memory request is the same as in Figure 4. The same set of experiments is performed in this case, as well. As we can see in the Figure 5 with the change of system size, the behavior of the system with respect to slowdown of individual applications and dynamicity of the system remains the same.

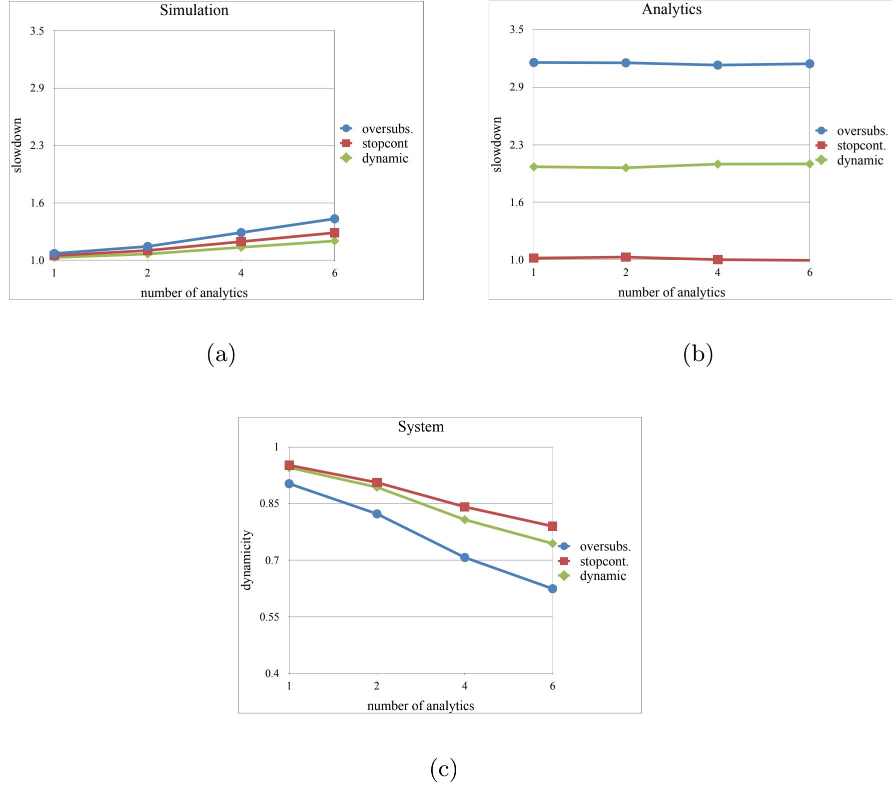
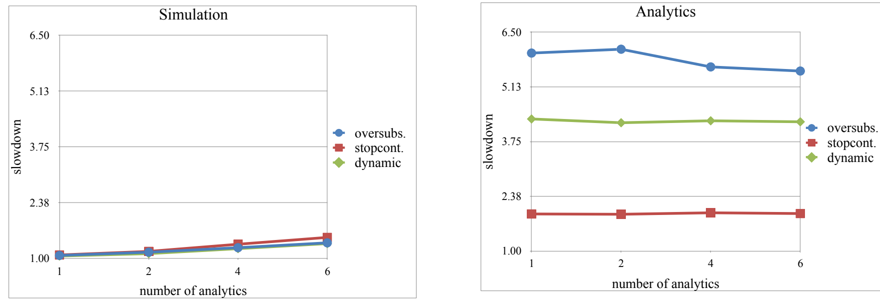
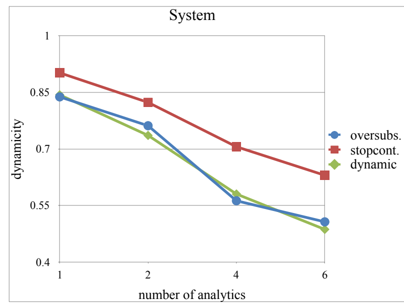


Fig. 3: Impact of number of analytics on: (a) slowdown of the simulation, (b) slowdown of the analytics, and (c) the dynamicity of the system. System size 1024 CPUs, i.e., 64 computing nodes. Total memory per node is 32GB. Simulation job requests 1024 CPUs. Analytics job requests 50% of simulation size. Total memory per node requested by simulation is *4GiB*. Total memory per node requested by analytics is 100MiB. Number of analytics per simulation run is indicated at x-axis.

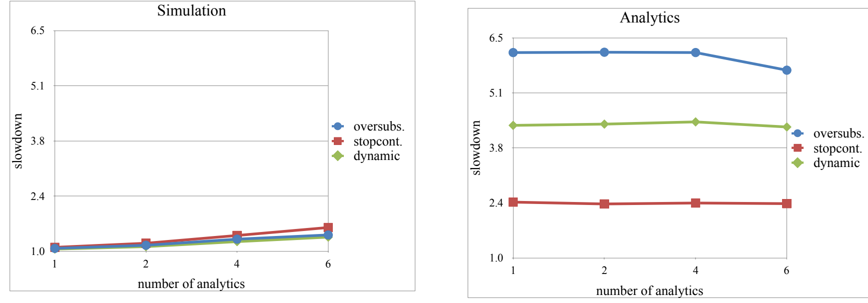


(a) (b)

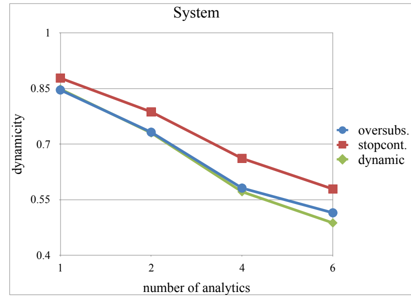


(c)

Fig. 4: Impact of number of analytics on: (a) slowdown of the simulation, (b) slowdown of the analytics, and (c) the dynamicity of the system. System size 1024 CPUs, i.e., 64 computing nodes. Total memory per node is 32GB. Simulation job requests 1024 CPUs. Analytics job requests 50% of simulation size. Total memory per node requested by simulation is *16GiB*. Total memory per node requested by analytics is 100MiB. Number of analytics per simulation run is indicated at x-axis.



(a) (b)



(c)

Fig. 5: Impact of number of analytics on: (a) slowdown of the simulation, (b) slowdown of the analytics, and (c) the dynamicity of the system. System size 512 CPUs, i.e., 32 computing nodes. Total memory per node is 32GB. Simulation job requests 512 CPUs. Analytics job requests 50% of simulation size. Total memory per node requested by simulation is *16GiB*. Total memory per node requested by analytics is 100MiB. Number of analytics per simulation run is indicated at x-axis.

5 Related work

DJSB benchmark, whose purpose is to evaluate the dynamicity of an HPC system, is a novel contribution in the literature, since it has never been tried to measure how the system reacts to dynamic adaptation of the workload.

Typical benchmarks tend to evaluate performance of a HPC systems by launching and measuring a set of applications' performance executed in isolation. Those performances are related to the application, that usually stresses only some component of the system, like processors, memory hierarchy and network. Some examples are the HPC Challenge Benchmark[1] or the benchmark used by Top500[5], Linpack[10].

The ESP Benchmark[19] is an approach for evaluating HPC performance. It evaluates system utilization and effectiveness by executing a medium length workload of 82 jobs, that varies in type of applications and requested resources, from a small job to jobs that take all system resources. With this approach, ESP permits to measure the efficiency of the system evaluating scheduling, and the system utilization. Our approach, DJSB permits, not only running diverse applications, but configuring differently the same applications in terms of memory requirements, application size, duration, etc. It evaluates the impact of different dynamic resource management approaches on each application individually, as well as the overall dynamicity of the system.

6 Conclusions and Future work

This paper presents DJSB, a tool targeted to evaluate HPC systems using different schedulers, applications characteristics and submission arguments. DJSB is an experiment-driven tool for HPC systems that, based on its configuration, executes a *reference stage* to collect reference performance metrics and later on executes the described workload. Workload description can be more or less specific, resulting in a fixed workload or something more variable. DJSB automatically collects performance metrics for applications and generates performance metric summaries and plots to make easy the comparison between scenarios. To illustrate the potential of DJSB, we have performed three different sets of experiments, each one including many variations concerning system size, application size, memory size, number of applications, etc. We have compared a stop&continue approach compared with oversubscription and a cooperative-dynamic resource management. Our experiments show that DJSB allows for an easy comparison of the systems that use different resource management approaches.

Acknowledgments. This work is supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (project TIN2015-65316-P), by the Generalitat de Catalunya (grant 2014-SGR-1051), by the European Union's Horizon 2020 research and innovation program under grant agreement No. 720270 (HBP SGA1).

References

1. Hpc challenge benchmark website, http://icl.cs.utk.edu/hpcc/
2. The human brain project, https://www.humanbrainproject.eu/
3. Marenostrum supercomputer, https://www.bsc.es/discover-bsc/the-centre/marenostrum
4. Message passing interface forum, http://www.mpi-forum.org/
5. Top500 website, https://www.top500.org/
6. Barcelona Supercomputing Center: The OmpSs Programming Model, https://pm.bsc.es/ompss
7. Clauss, C., Moschny, T., Eicker, N.: Dynamic process management with allocation-internal co-scheduling towards interactive supercomputing. In: Proc. 1th Workshop Co-Scheduling of HPC Applicat.(Jan 2016) (2016)
8. Dagum, L., Enon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE 5(1), 46–55 (1998)
9. Desai, N.: Cobalt: an open source platform for hpc system software research. In: Edinburgh BG/L System Software Workshop (2005)
10. Dongarra, J.J., Luszczek, P., Petitet, A.: The linpack benchmark: past, present and future. Concurrency and Computation: practice and experience 15(9), 803–820 (2003)
11. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)
12. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Malleable iterative mpi applications. Concurrency and Computation: Practice and Experience 21(3), 393–413 (2009)
13. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: The correct way to summarize benchmark results. Commun. ACM 29(3), 218–221 (1986)
14. Henderson, R.L.: Job scheduling under the portable batch system. In: Job scheduling strategies for parallel processing. pp. 279–294. Springer (1995)
15. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2015)
16. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (Dec 1995)
17. Smith, J.E.: Characterizing computer performance with a single number. Commun. ACM 31(3), 1202–1206 (1988)
18. Utrera, G., Tabik, S., Corbalan, J., Labarta, J.: A job scheduling approach for multi-core clusters based on virtual malleability. In: Euro-Par 2012 Parallel Processing, pp. 191–203. Springer (2012)
19. Wong, A.T., Oliker, L., Kramer, W.T., Kaltz, T.L., Bailey, D.H.: Esp: A system utilization benchmark. In: Supercomputing, ACM/IEEE 2000 Conference. pp. 15–15. IEEE (2000)
20. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 44–60. Springer (2003)
21. Zhou, S., Zheng, X., Wang, J., Delisle, P.: Utopia: a load sharing facility for large, heterogeneous distributed computer systems. Software: practice and Experience 23(12), 1305–1336 (1993)