Dynamic Process Management with Allocation-internal Co-Scheduling towards Interactive Supercomputing

Carsten Clauss
ParTec Cluster Competence Center GmbH, Munich
clauss@par-tec.com

Thomas Moschny
ParTec Cluster Competence Center GmbH, Munich
moschny@par-tec.com

Norbert Eicker
Jülich Supercomputing Centre
Forschungszentrum Jülich
n.eicker@fz-juelich.de

ABSTRACT

Heading towards exascale, the challenges for process management with respect to flexibility and efficiency grow accordingly. Running more than one application simultaneously on a node can be the solution for better resource utilization. However, we believe that this approach of co-scheduling can also be the way to go for gaining a degree of process malleability and dynamicity that can enable some kind of interactivity also in the domain of high-performance computing. In this paper, we present the recent advances made in this respect within ParaStation MPI, a high performance MPI library supplemented by a complete framework comprising a scalable and dynamic process manager. The paper presents four new scheduling policies, implemented in ParaStation MPI, for starting multiple MPI sessions concurrently and interactively within a single allocation of nodes. The features of these policies are detailed and evaluated by applying the Dynamic Job Scheduler Benchmark (djsb), a tool developed by the Barcelona Supercomputing Center especially for measuring interactivity and dynamicity metrics.

Keywords

Scheduling Policies, Co-Scheduling, Process Management, Interactive Supercomputing, High-Performance Computing

1. INTRODUCTION

Since the beginning of the pre-exascale era, there has been a rising demand for the support of interactivity and malleability also in the domain of high-performance computing. Such a support will allow supercomputer users to interact with their running applications, for example, in order to steer the progress of a simulation during runtime. It is widely believed that—besides some kind of a conceivable real-time interaction, for example, via graphical user interfaces for in-situ visualization—on large-scale supercomputers, such an interaction will primarily be conducted via additional applications to be started concurrently on the user's demand. (For proving this statement refer, for example, to the Technical Requirement Document [1] of Pre-Commercial Procurement (PCP) announcement issued by the Human Brain Project (HBP): In the context of the HBP it is foreseen to build up (pre-) exascale supercomputing systems featuring interactivity for large-scale brain simulations.[1])

According to this, each job will consist of multiple job steps (potentially divisible into primary and secondary ones) that may be launched interactively and that in turn can interact among each other. So, for instance, a user may run a large and long lasting simulation application, which then can interact during runtime with intermediately started auxiliary applications. Such secondary applications, which are then to be co-located with the primary application (either within its existing allocation or by requesting further resources) could then attach and interact with the long-running simulation in order to track and even govern its evolution. By co-locating the processes within the existing allocation, the linked applications can then take advantage especially of data and communication locality. Conceivable use case scenarios are, for example, visualization pipelines and the online post-processing of intermediate simulation steps as well as computational workflows and coupled codes for providing further input parameters during runtime of the primary simulation. Since such user interventions as well as the reactions made by the applications based on their interaction are not predictable, a dynamic and continuous sub-partitioning of the allocated resources is the consequence.

Such an allocation-internal co-scheduling may on the one hand aim at optimal system utilization. On the other hand, as user interactivity is also a matter of responsivity, the scheduling policy may focus on some kind of priorities. At this point, the above-mentioned demand for job malleability comes into play: Such a malleability comprises the question of the actual starting order of concurrently launched MPI sessions, the related question of a dynamic process-to-core assignment, the demand for the ability to reduce or increase the number of cores devoted to a certain MPI session, and the request for the possibility to suspend a whole MPI session on a temporary basis. Although these aspects are in the first instance relevant to the job-internal process management, at least the issue of reducing or increasing the number of processes and/or cores of an MPI session may also involve the system's higher-level resource manager.

For clarifying the different terms used in this paper, Figure 1 should illustrate the hierarchy of entities that have to be taken into account for the overall resource manage-

[1] www.humanbrainproject.eu

COSH 2016 Jan 19, 2016, Prague, CZ
© 2016, All rights owned by authors. Published in the TUM library.
DOI: 10.14459/2016md1288835

ment: The whole system is usually a *cluster* composed of *nodes*, while each node commonly features multiple *cores* resp. hardware threads. The user can request for a set of nodes/cores in terms of a *job allocation* for starting multiple parallel *applications* in terms of concurrent MPI *sessions* within.

Taken as a whole, the comprehensive management system then forms a three-tier hierarchy: At node level, the Linux scheduler manages the processes and threads, potentially governed by a predefined process pinning scheme. At job level, the local process manager handles the process-to-node/core assignment by starting, controlling and monitoring parallel MPI processes within the allocation devoted to the respective job. Finally, at cluster level, an outer resource manager maintains the different job queues of the batch system and performs the overall resource assignments granted by a job scheduler.

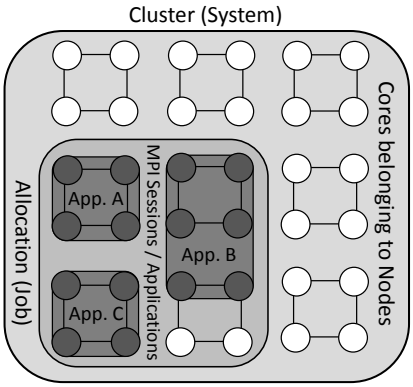

Figure 1: Naming of tiers in the system hierarchy

2. PARASTATION MPI

ParaStation MPI is an open-source MPI library, developed and supported by ParTec GmbH[2] under the umbrella of the ParaStation Consortium—an alliance consisting of ParTec, the University of Wuppertal, the Karlsruhe Institute of Technology (KIT), and the Jülich Supercomputing Centre (JSC). This MPI library has already proven to scale very well on large parallel production systems: In June 2009, ParaStation pushed the 3,288 node JuRoPA cluster at the JSC with more than 25,000 MPI processes to No. 10 in the world according to the Top 500 list. Furthermore, its successor, the JURECA system at the JSC with 49,476 cores, has just recently entered the November 2015 list at No. 50—again propelled by ParaStation [2].

The ParaStation MPI library (psmpi) is embedded into a complete framework for providing state-of-the-art cluster-based supercomputing [3]: Besides a robust and efficient cluster middleware, the ParaStation software suite also comprises sophisticated administration components like the ParaStation *ClusterTools* (for provisioning and maintenance), the ParaStation *HealthChecker* (for automated error detection and integrity checking) and the ParaStation *TicketSuite* (for analyzing and keeping track of issues).

psmpi is fully MPI-3 compliant [5], including support for the recent additions to the RMA interface, and also supports the MPICH-related Process Manager Interface (PMI) [4] as well as MPI-2 compliant dynamic process spawning—a feature long time neglected by other MPI implementations, but efficiently used by psmpi in the context of the Dynamical Exascale Entry Platform (DEEP) project [6], a project funded from 2011 to 2015 by the EU 7th Framework Programme.[3]

2.1 The Process Management System

The management facility of psmpi, called ParaStation Management (psmgmt), offers a complete process management system that can in turn be combined with an outer and more generic resource manager together with a batch queuing system plus job scheduler like TORQUE/MAUI or SLURM. The process management of psmgmt includes the creation of processes on remote nodes, control of the I/O channels of the remotely started processes, and the management of signals across node boundaries.

Since psmgmt knows about the dependencies between the processes and threads building a parallel session on a number of nodes of the cluster, it is able to take them respectively into account. That way, processes are no longer independent but form an entity in the same sense as the nodes are no longer independent computers but form a cluster of nodes as a self-contained system. This feature of psmgmt for handling distributed processes as a single unit plays an important role especially in the context of job control and allocation-internal scheduling—as it will be detailed later in this paper.

One important key to ParaStation's scalability is its efficient communication subsystem for inter-daemon messages. This subsystem, which uses the implementation of a highly-scalable Reliable Datagram Protocol (RDP), is used for resource monitoring as well as for launching and controlling the parallel processes by means of a network of ParaStation Daemons (psid). So, for example, this subsystem also performs process pinning, I/O forwarding and signal handling, and it ensures a proper resource cleanup after job termination.

2.2 Relation to the Resource Manager

Following a *one daemon per cluster node* concept, the daemon architecture is kept such generic that it can easily be extended by plugins for consolidating various services. Furthermore, this network of daemons also enables third-party services to piggyback their payload on the ParaStation communication subsystem. So, for instance, a TORQUE-related plugin (psmom) and a SLURM-related plugin (psslurm) efficiently replace the native daemons of these resource managers on the compute nodes in a ParaStation environment.

Figure 2 illustrates the orchestration between psmgmt and SLURM, as it is currently employed on the JURECA system at the Jülich Supercomputing Centre. As one can see, the psid together with its psslurm plugin plays the central role regarding process startup and job control on the compute nodes. SLURM itself is designed to operate even in heterogeneous clusters with up to tens of millions of processors and can accept thousands of job submissions per second with a sustained throughput rate of hundreds of thousand jobs per hour. Its direct linkage on JURECA to the network of distributed psids makes this orchestration between SLURM and ParaStation highly scalable and very flexible.

However, in case that the number of computing resources should actually be increased by MPI-2 compliant dynamic

[2] www.par-tec.com

[3] www.deep-project.eu

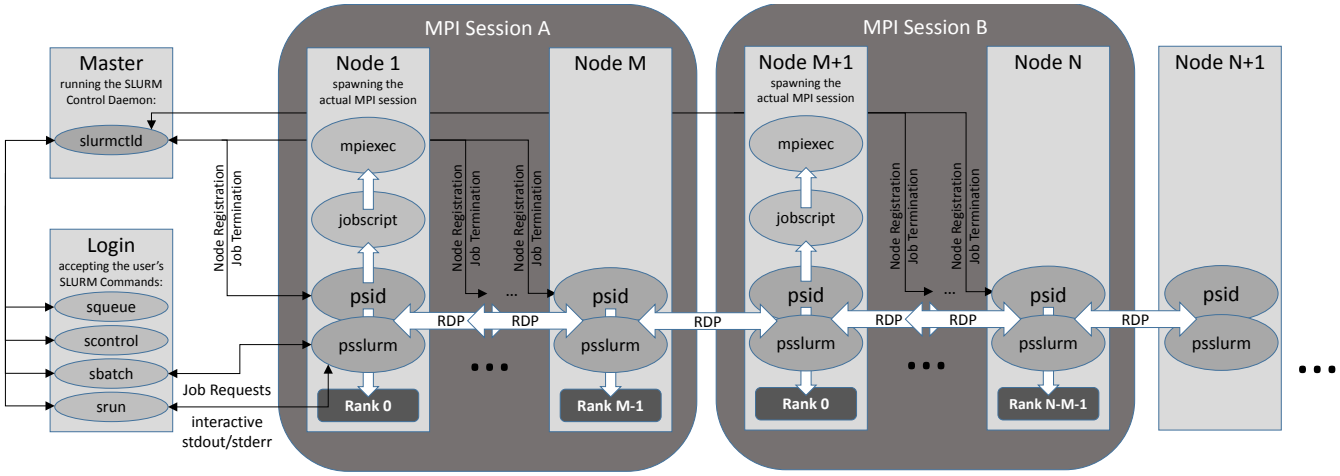

Figure 2: Orchestration between SLURM and ParaStation Management (psmgmt) via its psslurm plugin

process spawning during a job's runtime, the MPI layer would need the capability of requesting such a *post-allocation* of nodes from the resource management system. Such a request has then to be negotiated between the entities, but may very well also be rejected if the requested resources are currently not available. In fact, this feature has recently also been added to the ParaStation environment in the context of the above-mentioned DEEP project: The TORQUE server, which is used together with the MAUI scheduler in the DEEP project, has been enhanced by facilities to receive, queue and process new resource requests by applications via the ParaStation daemon subsystem during the runtime of a job. Moreover, the porting of these features to a SLURM environment, where the psslurm plugin for the psid and an additional plugin for the SLURM resource manager will jointly accomplish these tasks, is currently on-going work.

3. DYNAMIC PROCESS MANAGEMENT

As already stated in the introduction, dynamic process management at job level targets the malleability of MPI processes within an existing job allocation represented by a certain set of nodes currently assigned to a particular user or user group. A first degree of interaction between the user and concurrently running MPI sessions can be achieved by sending operating system signals to and between the respective parallel processes. However, such signals, as they are supported by all customary operating systems, are normally only valid in the context of the local node the operating system is running on. Hence, for supporting signal forwarding even across node borders, the respective middleware—thus, in our case, the local process manager—has to be capable of such a distributed signal handling.

3.1 Signal Handling and Process Pinning

In fact, psmgmt is already capable of doing so and natively takes care for the handling of process signals in a cluster-wide manner. This being so, some kind of runtime interactivity between the user and the started MPI sessions is already possible in this way. So, for example, sending a common `SIGTSTP` to ParaStation's `mpiexec` command would cause the whole respective MPI session to get suspended, whereas a `SIGCONT` can be issued to resume it later on.

Moreover, instead of a complete job preemption, even a temporal reduction of computing resources devoted to an MPI application is possible. According to this approach, the psids get signaled to perform a runtime adjustment of the rank-to-core pinning on each of the job's compute nodes. By means of such a re-pinning, some processes of a given job are moved within the respective nodes in such a manner that an appropriate fraction of oversubscribing for a certain group of cores is achieved.

The main advantage of this approach is that it is still transparent to the application. However, the temporal oversubscription will induce an operating system related scheduling overhead that might disturb the application's internal load-balancing scheme. Therefore, this method of re-pinning and oversubscribing should rather be a temporary measure in order to clear space, for example, for a short-running auxiliary application that is to be attached to a long-running simulation.

Based on these two approaches (suspend/resume and re-pinning of processes), new features for realizing job malleability and interactivity have been implemented recently within ParaStation. According to these approaches, the user can (for example, in context of an interactive SLURM session) launch multiple MPI applications in parallel and/or subsequently. As long as there are enough slots within the current allocation (according to the terminology of ParaStation, these *slots* are the hardware threads that can be assigned to processes resp. software threads), the psid will ensure via pinning that all slots are used exclusively by the assigned processes and threads. However, if the available slots get exhausted, allocation-internal scheduling policies come into play.

Moreover, the same applies to the case of MPI-2 compliant dynamic process spawning if a post-allocation of further nodes gets rejected by the resource manager. Although the initial started number of MPI ranks (this is the initial *world* size) may intentionally be smaller than the number of available slots within the allocation (this is the current *universe* size), hence leaving space in terms of free slots where new MPI processes can be spawned to, if all slots become populated, an oversubscribing or some other allocation-internal scheduling policy has to be applied for further spawn calls.

3.2 Allocation-internal Scheduling Policies

Currently, four of such policies for job- or allocation-internal co-scheduling are implemented in psmgmt: One that just lets the newly started processes wait for getting free slots, one that simply voids the previous exclusiveness of resources and thus allows for oversubscribing of slots, one that follows the suspend/resume approach in such manner that each subsequently started MPI session suspends its respective predecessor for getting free slots, and one that uses re-pinning for a temporal reduction of computing resources concerning the still running predecessor of the newly started application. Moreover, for most of these approaches further sub-policies are conceivable and to some extent already implemented in psmgmt.

All the four policies, as they are detailed in the following paragraphs, can currently be used by means of a wrapper script called `psmpiexec` that extends the common `mpiexec` command as commonly provided by psmpi. However, at this point it should be emphasized that both commands are more or less just user interfaces that can easily be replaced by others—so, for example, by a more SLURM-like `srun` frontend.

The Wait Policy.

According to this policy, any newly started MPI session that can no longer be scheduled into free slots has to wait until one or more of the previous ones gets finished so that enough slots become available again. As this policy still sticks to the original paradigm of preventing any oversubscription, mutual interferences between the sessions should almost be avoided. However, on the other hand, if the interaction between the sessions demand for a concurrent execution, this policy cannot safely be used.

The Surpass Policy.

This policy is based on the suspend/resume mechanism of psmgmt: Every time a new session gets launched within an allocation with already filled slots, the prior session(s) (these are the ones issued by preceding `psmpiexec` within the same allocation) get(s) automatically suspended until the successor becomes finished. The idea behind this policy is that the most recently started session should most probably be the one with the highest priority from the user's point of view. According to this idea, the user can start further sessions (for example for short running auxiliary applications) that then will surpass previously started, long-running ones.

The Overbook Policy.

When this policy is enabled and all free slots are exhausted, all the MPI sessions are run concurrently and in a competitive manner on the nodes and cores of the allocation. The question whether there should still be some kind of a pinning scheme in such an overbooked situation, or if all the processes should then be enabled to flow freely across the cores of their respective nodes, could be then considered as a further sub-policy.

The Sidestep Policy.

This policy is quite similar to the overbook policy. However, the difference is that here the processes of the already running applications are re-pinned in such a manner that the processes of the new session run on their cores exclusively. That means that, in a first instance, only those cores are overbooked where the processes of the preceding MPI sessions are pinned to.

The Spread Option.

Normally, ParaStation places all processes as compactly as possible (with due regard to any threads) onto the nodes. However, in cases where a small number of newly started processes are overloading an allocation already filled up with running applications, it might be beneficial to have the processes of the new session get started on the nodes as widespread as possible. This can be achieved by using an additional *spread* option, which is therefore meaningful together with the *overbook* or the *Sidestep* policy. Using the spread option, the hope is that the already running MPI sessions will not get as much affected by the additionally started processes as it would be the case if the latter were all started on one (or only a few) node(s) of the allocation.

4. EVALUATION OF THE POLICIES

The Dynamic Job Scheduler Benchmark (dsjb) is a tool developed by the Barcelona Supercomputing Center (BSC) for evaluation different scheduling solutions. Although its description[4] as well as its source code[5] are publicly available, this section initially gives a more detailed introduction into the respective benchmarking metric since the knowledge about this seems up to now not very widespread. After this introduction, this section presents some early results gained by applying this benchmark together with the new allocation-internal scheduling policies of psmgmt as detailed in Section 3.2.

4.1 Benchmark Description

The djsb has originally been written and released in the context of the Pre-Commercial Procurement (PCP) of the Human Brain Project (HBP). Its primary purpose is to allow for a performance comparison of the different resource management solutions proposed by different tenders during the PCP. In doing so, the benchmark focuses on *interactivity* and the *dynamicity* of the proposed job scheduling systems. The benchmark actually consists of multiple processes and threads performing the STREAM benchmark [8] in parallel—hence without any considerable communication.

The idea of this benchmark is to let two synthetic applications run concurrently within the same job allocation: one longer running "*simulation*" application and one shorter running "*analysis*" application, both to be modeled by the STREAM executable. The benchmark basically measures the runtime of each of both when they are started separately, as well as the runtime when they are executed concurrently. Based on these durations, the benchmark calculates some reasonable efficiency numbers (the so-called *Simulation/Analysis/Wait Coefficients*) and finally reports a *Dynamicity Ratio* as a product of those three coefficients.

Although the djsb focuses on interactivity, the synthetic applications are issued as MPI sessions automatically by a Python-based benchmarking script that models the hypothetical user of the job allocation by sporadically calling `mpiexec` for the short-running analysis application. Please note here that the document officially describing the bench-

[4] http://pm.bsc.es/~vlopez/files/djsb_doc.pdf
[5] http://pm.bsc.es/~vlopez/files/djsb.tar.gz

mark [7] uses a different nomenclature than we do within this paper: In the official djsb description, the term *job* refers to a single application (rather than to an allocation) and hence to the term of an MPI *session* according to the terminology used in this paper.

The actually measured parameters and performance metrics of the djsb are:

- *Wait Time:* Time that has passed between the session request issued by the "user" (this is the call of `mpiexec`) and the actual start of the respective application.
- *Execution Time:* Time that has passed between session start and its completion. This is hence the effective runtime of the application.
- *Response Time:* Time that has passed between session request by the "user" and its completion. This is hence the sum of Wait and Execution Time.
- *Slowdown:* Performance decrease in terms of the ratio between the actually measured times and the reference scenario where all sessions are run consecutively.
- *Expected vs. Real Slowdown:* While the Expected Slowdown is a pre-calculated value based on theoretical assumptions, the Real Slowdown is the actually observed one.
- *Efficiency Coefficient:* This is just the ratio of Expected and Real Slowdown. Higher values are better.

$$E = \frac{\text{Expected Slowdown}}{\text{Real Slowdown}}$$

- *The Wait Coefficient:* This value is calculated according to the following formula. (Please refer to the official djsb description [7] for a more detailed explanation of this.) Values close to or even greater than 1 are better.

$$W = \frac{\text{Wait Time in static case} + \text{Normalization Constant}}{\text{Wait Time in dynamic case} + \text{Normalization Constant}}$$

- *Dynamicity Ratio:* This is the product of the Efficiency Coefficients as measured for both synthetic applications (the long-running simulation and potentially several short-running analysis sessions)

$$D = E^{\,simulation} \cdot E^{\,analysis} \cdot W^{\,analysis}$$

4.2 Measured Benchmark Results

The benchmark results presented in this section were all obtained on an allocation with 4 nodes and 160 cores in total. The process/thread configuration chosen was as follows for all the benchmark runs:

	no threads	*with threads*
Simulation application	160 procs (40 per node)	32 procs (8 per node, 5 threads per proc)
Analysis application	32 procs (all on one node, or 8 per node with *spread* option)	8 procs x 4 threads (all on one node, no *spread* option used)

Without using the *spread* option, as detailed in Section 3.2, the chosen configuration would overbook the first node of the allocation with 32 analysis processes. In contrast, if the *spread* option is enabled, the 32 processes will be distributed across all 4 nodes of the allocation so that each node would then be overbooked by "only" 8 processes.

The following paragraphs show and briefly discuss the single coefficients and the overall dynamicity results that have been measured with this configuration for the different scheduling policies:

The Wait Policy.

	no threads	*with threads*
Simulation Efficiency:	1.04	1.03
Analysis Efficiency:	2.0	2.03
Wait Coefficient:	0.55	0.55
Dynamicity Ratio:	**1.15**	**1.15**

Since both applications are run within the allocation separately according to this policy, their efficiency (and hence their runtime seen individually) are quite good but the overall Wait Coefficient is relatively bad due to the long waiting time of the analysis application before it gets started. However, the overall measured Dynamicity Ratio is here greater than 1, what means that this policy improves the dynamicity and hence the anticipated capability for interactivity—at least with respect to the metric of the djsb benchmark.

The Surpass Policy.

	no threads	*with threads*
Simulation Efficiency:	0.86	0.86
Analysis Efficiency:	2.0	2.0
Wait Coefficient:	1.0	1.0
Dynamicity Ratio:	**1.72**	**1.73**

The main advantage of this policy is that long pending times of the analysis applications are avoided and that at least the efficiency of the analysis sessions should (and is) as good as in the case of the Wait policy. However, as the simulation application gets completely interrupted, its duration gets extended accordingly so that its efficiency is decreased in comparison to the Wait policy. Moreover, since the analysis sessions are usually not only shorter in runtime, but also smaller in the number of processors used, both policies (Wait and Surpass) may lead to a temporary under-utilization of the allocation.

The Overbook Policy.

	no threads	*with threads*	*spread option (no threads)*
Simulation Efficiency:	0.87	0.91	1.02
Analysis Efficiency:	1.23	0.89	1.02
Wait Coefficient:	1.0	1.0	1.0
Dynamicity Ratio:	**1.07**	**0.81**	**1.04**

In the case of this policy, the simulation as well as the analysis are run concurrently and in a competitive manner within the allocation. This usually means that the processes of the analysis sessions get started (and pinned) onto a subset of those cores where the simulation processes are already running on. Although this policy guarantees that there are no unnecessary idle times of cores during the benchmark's run,

the efficiencies of both the simulation application as well as the analysis application are liable to get impaired due to the temporary overload.

The Sidestep Policy.

	no threads	*with threads*	*spread option (no threads)*
Simulation Efficiency:	0.87	0.87	1.34
Analysis Efficiency:	1.61	1.5	1.13
Wait Coefficient:	1.0	1.0	1.0
Dynamicity Ratio:	**1.41**	**1.3**	**1.5**

These results show that this policy gains a very good Dynamicity Ratio, but when looking at all four policies it becomes clear that the Surpass policy gains the best results. However, it has to be emphasized, that especially the Surpass policy does not allow for an MPI-based interaction between both sessions via message-exchange due to the fact that the simulation session actually gets suspended during the runtime of the analysis application.

In fact, most of the efficiency coefficients are usually expected (at least in theory) to be in the range of [0,1] because this would represent the case when the applications share some of their resources at some point in time. On the other hand, the coefficients are greater than 1 when an application runs with more resources than expected—like in a session serialization.

However, according to the benchmark results we have measured and presented here, all four of the new allocation-internal scheduling policies would improve the dynamic behavior and thus the capability for interactivity. All in all, this indicates two facts for us:

1st: The metric of the djsb benchmark (here especially the calculation of the Expected Slowdown as well as the applying of some "magic" Normalization Constants) seems to be not very well balanced for all scenarios. However, it has to be emphasized that for the HBP-PCP, the benchmark scenarios are well-defined and differ from the process/thread configuration used for our measurements—it is most likely that the internal benchmarking parameters of the djsb are tailored to those configurations as given by the HBP-PCP.

2nd: Since the djsb totally neglects the actually required message-exchange between the concurrent sessions, the benchmark can only give a first hint for the interactive behavior of a system, but cannot really judge about complex interactivity scenarios as they are envisaged for future supercomputing systems. On the other hand, it is in the nature of things that benchmark scenarios have to tend to simplify things in order to make their results more conferrable.

5. CONCLUSION AND OUTLOOK

In this paper, we have presented recent advances made for ParaStation MPI (psmpi) and its process manger (psmgmt) with respect to co-scheduling and process malleability at job level. While co-scheduling is frequently associated with a means for better resource utilization, the approaches presented in this paper are primarily not so much *resource*-centric but rather *user*-centric, as they focus on *interactivity*. In doing so, four new policies for scheduling of concurrent MPI sessions within a single interactive job allocation have been presented and evaluated by means of the Dynamic Job Scheduler Benchmark (djsb). It turned out that (at least according to the metric used by the djsb) all four new policies help to improve the desired scheduling behavior towards malleability and interactivity. However, at the same time it became clear that the results of this benchmark are not quite meaningful when it comes to how concurrent sessions can actually interact between each other because the omits any communication metrics.

All in all, we believe that interactivity will become more and more important also in the domain of supercomputing and that a dynamic and malleable process management, as presented in this paper, is the first right step towards this challenge.

6. REFERENCES

[1] *HBP-PCP Technical Requirements concerning the R&D services on "Whole System Design for Interactive Supercomputing"*, Forschungszentrum Jülich, Human Brain Project, April 2014, online available: http://apps.fz-juelich.de/hbp-pcp/

[2] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer: www.top500.org – *Lists of World's TOP500 Supercomputers*, November 2015.

[3] ParTec Cluster Competence Center GmbH: *ParaStation Cluster Suite – product portfolio*, online available: http://www.par-tec.com/products/overview.html

[4] Pavan Balaji et al.: *PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems*, in Recent Advances in the Message Passing Interface: 17th European MPI User's Group Meeting, Springer Lecture Notes in Computer Science, pages 31–41, September 2010.

[5] The Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard – version MPI 3.1*, printed at the High Performance Computing Center Stuttgart (HLRS), June 2015.

[6] Norbert Eicker, Thomas Lippert, Thomas Moschny, and Estela Suarez: *The DEEP Project – Pursuing Cluster-Computing in the Many-Core Era*, in Proceedings of the 42nd International Conference on Parallel Processing (ICPP), IEEE Computer Society Press, pages 885–892, October 2013.

[7] Marcal Sola and Victor Lopez: *Dynamic Job Scheduler Benchmark – HBP-PCP Benchmark Documentation*, July 2014, online available: http://pm.bsc.es/~vlopez/files/djsb_doc.pdf

[8] John D. McCalpin: *Memory Bandwidth and Machine Balance in Current High Performance Computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[9] Suraj Prabhakaran et al.: *A Batch System with Fair Scheduling for Evolving Applications*, in Proceedings of the 43rd International Conference on Parallel Processing (ICPP), IEEE Computer Society Press, pages 351–360, September 2014.

[10] Suraj Prabhakaran et al.: *A Batch System with Efficient Scheduling for Malleable and Evolving Applications*, in Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society Press, pages 429–438, May 2015.