An Experimental Environment for the Evaluation of Exascale HPC Runtimes

Joseph Emeras, Emmanuel Jeanvoine, Lucas Nussbaum

An Experimental Environment for the Evaluation of Exascale HPC Runtimes

Joseph Emeras, Emmanuel Jeanvoine, Lucas Nussbaum

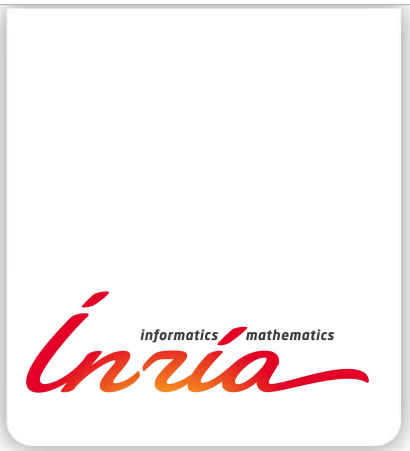
RESEARCH REPORT

N° 8482

February 2014

Project-Team Algorille

ISSN 0249-6399 ISRN INRIA/RR--8482--FR+ENG



An Experimental Environment for the Evaluation of Exascale HPC Runtimes

Joseph Emeras, Emmanuel Jeanvoine, Lucas Nussbaum

Project-Team Algorille

Research Report n° 8482 — February 2014 — 13 pages

Abstract: The era of Exascale computing raises new challenges for HPC. Intrinsic characteristics of those extreme scale platforms bring energy and reliability issues. To cope with those constraints, applications will have to be more flexible in order to deal with platform geometry evolutions and unavoidable failures. Thus, to prepare for this upcoming era, a strong effort must be made on improving the HPC software stack. This work focuses on improving the study of a central part of the software stack, the HPC runtimes. To this end we propose a tool that aims at evaluating such runtimes, at scale, to enable the discovery of performance gaps, understand future bottlenecks, and evaluate fault tolerance and load balancing mechanisms. Extensive experimentation showing the benefits of our approach has been performed with two HPC runtimes: Charm++ and OpenMPI.

Key-words: experimentation, HPC runtimes, large scale, fault tolerance, emulation.

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Un environment expérimental pour l'évaluation des runtimes HPC Exascale

Résumé : L'arrivée des plates-formes Exascale pose de nouveaux problèmes pour le HPC. Les charactéristiques intrinsèques à ces plates-formes de taille extrême génèrent des problèmes de fiabilité ainsi que des problèmes énergétiques. Pour faire face à ces types de contraintes, les applications vont devoir devenir plus flexibles et s'adapter aux évolutions de la géométrie de la plate-forme ainsi qu'aux pannes, devenues inévitables. Ainsi, pour se préparer à l'Exascale, un effort important doit être fait dans le but d'améliorer la pile logicielle du HPC. Ce travail se concentre sur l'amélioration et l'évaluation d'un aspect central de la pile logicielle du HPC: le runtime. Dans ce but, nous proposons un outil permettant d'évaluer de tels runtimes, à grande échelle, afin d'aider à la découverte de problèmes de performance, d'appréhender les futurs goulots d'étranglement aisni que d'évaluer les mécanismes de tolérence aux fautes et d'équilibrage de charge. Une expérimentation conséquente nous a permis d'étayer nos propos et d'illustrer les bénéfices de notre approche sur deux types de runtime HPC: Charm++ et OpenMPI.

Mots-clés : expérimentation, runtimes HPC, large échelle, tolérence aux fautes, émulation.

1 Introduction

During the last decade, numerical simulations have become a key component of most advances in today's society, science and engineering. Those simulations, computing interactions of many complex models, leverage the computing power of modern HPC infrastructures. The increasing scale and volume of those simulations is a driver for infrastructure evolution, and before 2020 HPC infrastructures will likely reach Exascale. Several projects around the world have studied the challenges of computing at Exascale and numerous improvements need to be achieved. In particular, The International Exascale Software Project [4] explored those challenges from the software stack point of view. One identified direction is the improvement of the HPC runtimes to ease the programming of Exascale applications. Among other directions, the report recommends to focus on the support of platform heterogeneity, load balancing, and resilience.

Evaluating current and future HPC runtimes is critical to detect potential defects or performance issues that may appear on Exascale platforms. As those platforms do not exist yet, experimental evaluation of the runtimes is very challenging.

This work contributes to enabling experimental evaluation of Exascale HPC runtimes. Starting from Distem, which is a versatile emulator for studying distributed systems, we designed an emulator suitable for the evaluation of HPC runtimes, enabling specifically: (1) emulation of a very large scale platform on top of a regular cluster; (2) introduction of heterogeneity and dynamic imbalance among the computing resources; (3) introduction of failures. Those features provide runtime designers with the ability to experiment their prototypes under a large range of conditions, which is necessary to anticipate the future needs of Exascale computing.

This paper is organized as follows. First, Section 2 provides an overview of Distem's design, and describes how it was extended to address the specific challenges of the evaluation of Exascale runtimes. Then, Section 3 presents three experiments that demonstrate the benefit of such a tool when dealing with HPC runtime evaluation. Section 4 compares our approach to other related works, and Section 5 concludes this paper and discusses our future work.

2 Design of an emulator to study Exascale HPC runtimes

After providing an overview of the design of Distem (Section 2.1), this section describes how Distem was extended to support large scale experiments (Section 2.2) and experimentation under changing conditions, including the presence of failures (Section 2.3).

2.1 Overview of Distem

The features, general architecture and evaluation of Distem were presented in [17], but are shortly summarized here for completeness. Distem is able to emulate a heterogeneous virtual platform on top of a homogeneous cluster in order to meet the experimenter's requirements. A virtual platform is composed of virtual nodes and virtual networks.

On the networking side, Distem can emulate a complete and complex network topology. Virtual nodes can belong to different virtual networks and can act as gateways between several networks. Network links can be configured in order to provide specific performance (latency, available bandwidth).

On the computing side, the computing power of the virtual nodes can be altered in order to emulate slower nodes in the virtual platform.

Distem also proposes two operating modes. In the real-scale mode, the total number of emulated resources will be the same as the number of physical resources on the platform. In the

node-folding mode, Distem uses *lightweight virtualization* to launch several virtual nodes (*vnodes* in Distem's terminology) on the same physical resource, that will be shared among them.

2.2 Supporting large scale experiments

Being able to deploy large scale virtual platforms is really important for several reasons. Designing the future Exascale runtimes or applications requires evaluation on large scale infrastructure. Exascale infrastructures do not yet exist and it is not always easy to get access to the current Petascale infrastructures, in particular for research purposes. Our approach, described in [17], leverages lightweight virtualization to perform high-density node folding.

Since [17], we have been focusing on large scale virtual platform deployment with Distem. Two aspects are specifically challenging: the deployment time and the network issues caused by the limited scalability of the ARP protocol.

2.2.1 Reducing the deployment time using vectorized operations

Originally, in Distem, every operation (like virtual network creation, *vnode* creation, *vnode* start, *vnode* stop) was performed by sending a request to a physical node that acts as a coordinator for the other physical nodes. After receiving a request, this coordinator node forwards the request to the physical node responsible for the related *vnodes*. This design suffers from performance and reliability issues when dealing with thousands of *vnodes*. To improve scalability, we added a set of vectorized functions for each of the previously listed operations. This works in two steps: (1) a single request is sent to the coordinator node; (2) the coordinator sends a request in parallel to all the physical nodes involved in the operation. With this optimization, a complex virtual platform with 10,000 *vnodes* can be deployed in a few minutes.

2.2.2 Overcoming the limited scalability of the ARP protocol

A common issue when running thousands of nodes in a single Ethernet network is the poor scalability of the ARP protocol, as it was already pointed out in other efforts to build large-scale virtualized environments [12]. Indeed, when a node `A` tries to contact another node `B` for the first time, an ARP request is sent by `A` to learn the MAC address of `B`. First, this mechanism is slow and might lead to timeouts when several thousands of nodes perform simultaneously the same kind of requests. Moreover, ARP tables that aim at storing the mapping between an IP and a MAC address might be overloaded. ARP tables exist in both the operating system of the *vnodes* and in the network equipment.

Two improvements have been added to Distem to overcome those issues. The first improvement consists in tuning the Linux kernel settings controlling the ARP tables of the virtual nodes, such as increasing the default size of the ARP tables (in order to never overload the tables, which would lead to removal of some entries) and increasing the aging time of the entries (in order to never have to remove entries from the tables). The second one consists in filling automatically the ARP tables of all the virtual nodes with the complete ARP$\leftrightarrow$ IP mapping of the *vnodes*. Thanks to this improvement, no future ARP request has to be performed.

2.3 Supporting heterogeneous and evolving experimental conditions and fault injection

Target applications for future Exascale infrastructures will have to deal with heterogeneous and evolving computing and networking capabilities. For instance, energy or over-heating concerns

may lead to reducing the CPU frequency/voltage of some resources, and hardware failures might require the shutdown of some resources.

2.3.1 Evolving virtual platform

We have previously shown [17] that Distem can be used to create virtual heterogeneous platforms from a homogeneous cluster, with *vnodes* with different frequencies and number of cores, and network links with different bandwidth and latency. Other presented features are also interesting to emulate failures. Three kinds of failure can be introduced in the virtual platform: (1) *vnode* shutdown; (2) network disruption such as a congested network link; (3) freeze of all processes of a *vnode*.

The virtual platform description (CPU frequency modification, network link modification, and virtual node arrival/disappearance) can be updated dynamically and upon request during an experiment.

2.3.2 Framework for event injection

Introducing a platform modification manually is convenient to test a given feature but does not allow complex and realistic experimentation. Thus, being able to modify the virtual platform automatically and in a deterministic way during an experiment is required. For that purpose, we added an event injection framework.

The following automatic modifications are supported for a given set of *vnodes*: (1) modification of the CPU frequency; (2) modification of the network interfaces capabilities (latency and bandwidth); (3) start and stop; (4) freeze and unfreeze of the processes.

The events can be specified in two ways. First, it is possible to use an event trace that specifies which modification occurs at which date (relatively to the start of the experiment). Second, it is possible to define automatically the date of event arrival according to various probability distributions. Currently, uniform, exponential and Weibull distributions are supported.

Thanks to this framework, the orchestration of complex and realistic platform modifications can be performed in order to achieve advanced experiments in a great variety of contexts.

Furthermore, it is a keystone for repeatable experiments. Several runs can be performed under the same conditions to ensure validity of the measurements (e.g., meaningful computation of mean values) and produce trustworthy scientific results.

3 Experimental evaluation

This section illustrates the benefits of Distem in the context of HPC runtime evaluation. For that purpose we performed three sets of experiments. First, we demonstrate the interest of resource virtualization for scalability evaluation (Section 3.1). Then, we show how Distem's failure emulation can be used to evaluate fault tolerance strategies under different MTBF values (Section 3.2). Last, we use Distem's platform heterogeneity emulation and its ability to dynamically change performance of nodes to study the behavior of Charm++ load balancing in a static, and then dynamic heterogeneous platform (Section 3.3).

Experimental evaluation often suffers from poor repeatability [5], which is a major problem to trust the results. Using Distem alleviates this issue as emulated platforms can be reproduced identically between two experiments, even when performed on different physical platforms. For the sake of reproducibility we also provide our experiments' output logs and scripts to generate the figures. These are available at `https://gforge.inria.fr/frs/download.php/33368/exascale14.tar.gz`.

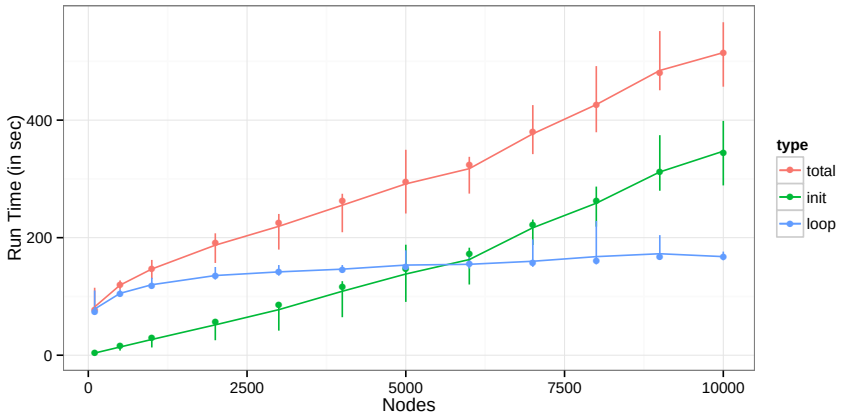


Figure 1: Experimentation of MPI collective operations scalability with Distem.

Experimental Setup

The experiments have been performed on two clusters of the Grid'5000 testbed [3], *Graphene* and *Griffon*, both from the Nancy site. *Graphene* is composed of 144 nodes (1 Intel X3440 CPU at 2.53 GHz, 4 cores/CPU, 16GB RAM each node) and *Griffon* is composed of 92 nodes (2 Intel L5420 CPU at 2.50GHz, 4 cores/CPU, 16GB RAM each node). Both clusters are interconnected with Gigabit Ethernet network and run a Debian Wheezy with a 3.2 Linux kernel.

3.1 Scalability evaluation with node folding

We propose a first experiment that demonstrates Distem's usefulness for scalability experimentation. For that purpose, we evaluate the scalability of collective MPI operations within a large scale platform emulated with Distem. We use a simple MPI benchmark that is composed of a loop of collective operations (*MPI_Bcast*, *MPI_Scatter* and *MPI_Gather*) executed 100 times. Data size in collective operations is insignificant compared to the number of messages sent. We measure the time taken to initialize the library, to process the loop and the total execution time. We use OpenMPI 1.4.5 and run the code with an increasing number of *vnodes*, from 100 to 10,000, hosted on up to 100 physical nodes, with a constant folding factor of 100 *vnodes* per physical node to ensure that results from different scales are comparable. One MPI process per *vnode* is executed. For each scale, we repeat the test at least 10 times.

Figure 1 presents the result of scalability study of the MPI benchmark. For each type of measure (MPI initialization, MPI collective operations loop and total), we plot the time taken as a function of the number of *vnodes* used.

In this figure, vertical bars represent the span of observed values at the corresponding scale. Dots correspond to the median values and we link the observations by the averages.

In this experiment, a large time-consuming part of the code is the initialization of MPI communication by rank 0 process (green curve). The same phenomenon of the poor scalability of *MPI_Init* has already been observed in [1]. As *MPI_Init* eagerly initiates all-to-all connections, it results in an initialization time that can take up to several minutes at very large scale. According to the authors of [1], to be scalable the MPI implementation should lazily establish a connection when a process needs to communicate with another.

However, we observe that MPI collective operations (blue curve) scale well with an increasing

number of nodes. Above 2000 *vnodes*, this time increases linearly and very slowly, despite collective operations putting a lot of stress on the runtime. This is promising for the use of MPI on Exascale platforms.

This experiment demonstrated Distem's ability, thanks to its node folding features, to evaluate scalability with up to 10,000 *vnodes* using only a limited number of physical nodes.

3.2 Evaluating fault tolerance strategies in Charm++

In this experiment, we use Distem to generate failures, and evaluate how the Charm++ middleware [7], with its support for fault tolerance, is able to cope with them.

We use the *Jacobi3D* application provided with Charm++, a 7-point 3-dimensional stencil code that implements in-memory checkpoint-restart [19]. With this mechanism, the Charm's runtime ensures that Charm++ objects on a node are replicated in memory on some other nodes at each checkpoint. To each node is assigned a set of *buddy* processors which will host its object copies. Upon node failure, at restart time the lost objects are retrieved from their copy on the remaining nodes. Then, Charm's runtime re-orchestrates their distribution to continue the computation. For this experiment, we use *Griffon* cluster and Charm++ compiled for the *net-linux-x86_64* architecture with the *syncft* option (activation of in-memory checkpoint-restart feature) and compiler optimization level 0, as higher optimization levels led to faulty restarts. We run 1000 iterations of *Jacobi3D* on 256 cores (32 nodes, 8 *vnodes* per node, 1 core per *vnode*) with a problem size of $2048 \times 2048 \times 1024$ and a decomposition of 256^3. We use Distem's real scale operating mode instead of node folding to avoid perturbing the computation.

We use Distem to inject *vnode* failures to observe Charm's ability to deal with different MTBF values. In particular, we use the event generator described in Section 2.3 and the Weibull distribution. This distribution is suitable for approximating MTBF behaviors [15, 9]. We use decreasing Weibull's scale parameter from 1 year (one of the classical MTBF values for one processor) to 1 day (to simulate faulty machines) and we use this event management feature on every *vnode* to ensure that they can fail independently and following the selected MTBF. We also choose a small shape parameter $k = 0.5$ (typical values being 0.5 and 0.7) to ensure that failures will arrive rapidly. With this, we increase the probability of encountering a failure in our experiments without changing the scale.

Figure 2 presents the results of the study of Charm's fault tolerance (FT) feature with different MTBF values. One can observe that the overhead of replication is quite low compared to the gain in case of failure. The replication only induces a slowdown of a few seconds on a 20 minutes run.

MTBF	**Iterations Lost at Failure #**						**Total**
	1	**2**	**3**	**4**	**5**	**6**	
365	40						40
180	140	110					250
90	50	20	120				190
60	30	0	60	70	140		300
30	50	0	0	30	80	60	220

Table 1: Number of iterations lost in failures.

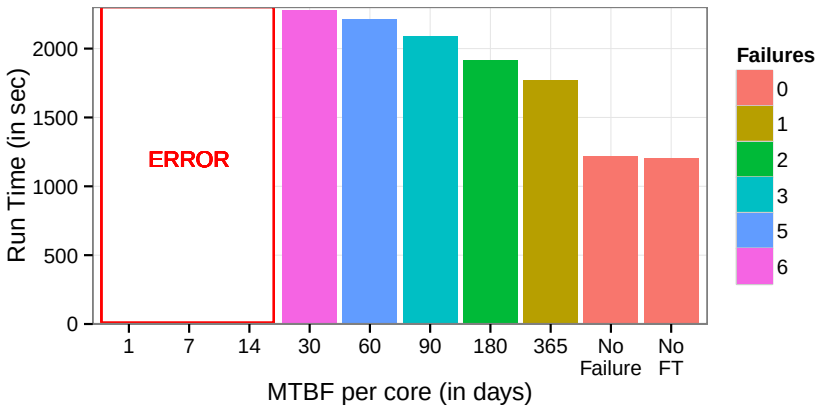


Figure 2: Charm's fault tolerance experimentation with Distem.

It is interesting to observe that a single failure (in the case of a 365 days MTBF) leads to a noticeable loss of time. When comparing with Table 1, which presents the number of iterations lost at a given failure[1], we see that for a 180 days MTBF, the total number of iterations lost is larger than for a 90 days MTBF but with a total run time being smaller. We measured that each process restart takes approximately 3 seconds. This highlights the fact that in this scenario, the loss of time in case of failure is mostly due to the detection of the failure itself and the preparation of the restart, that vary between 10 and 20 seconds. It is what happened in the aforementioned case for the 90 days MTBF value, where failure detection and restart preparation times were longer.

In this experiment, thanks to Distem's failure generation framework, we were able to easily simulate node failures with different MTBF values and thus to observe Charm's FT efficiency.

3.3 Evaluating load balancing strategies in Charm++

In this experiment we use Distem to generate heterogeneous conditions from a homogeneous platform and to generate dynamic perturbation on the nodes. With this setup, we illustrate Charm's load balancing (LB) behavior in heterogeneous conditions. We use the *Griffon* cluster (still with 1 core per *vnode*) and run the *Stencil3D* application. This code belongs to the same class of application as *Jacobi3D* but is simpler and does not implement all the Charm's FT object replication mechanism. This application has been chosen since it is relevant to observe the benefit of Charm's LB functionality.

For this experiment we use Charm++ compiled for the *net-linux-x86_64* architecture with optimization level 3. We run 100 iterations of Stencil3D with a 1024^3 problem size and a 64^3 decomposition on three kinds of experiment:

- A homogeneous platform without perturbation on the *vnodes*.
- A heterogeneous platform where half of the *vnodes* have a CPU clock reduced by Distem from 2.5GHz to 1.5GHz. This case is an interesting study of what would be the result of running Stencil3D on 2 cluster partitions with different CPU clock speeds.

[1] A loss of 0 iteration is due to 2 contiguous failures, in this case, the 2nd failure does not impact the number of iterations lost.

- A homogeneous platform where the available CPU power of a subpart of the *vnodes* is dynamic. Each two minutes, 1/8 of *vnodes* are downclocked by Distem for two minutes, then set back to their original frequency. This scenario can mimic a simulation of a cooling failure that induces a periodical downclocking of the nodes located in the cluster's hot spots. Another example of such a varying platform performance is a computation within a multi-tenancy environment [6].

For each experiment we run the application with and without Charm++ load balancing. Thus we test the following LB options:

- **LBOff**: load balancing is deactivated. Charm++ objects will not be migrated and no processor imbalance check will be done by Charm++ runtime.

- **RefineLB**: we use Charm's Refine load balancer. This centralized balancer moves objects away from the most overloaded processors to balance the workload. It also limits the number of objects being migrated.

Results are presented as *Projections* [8] CPU usage graphics, a performance analysis and visualization tool for Charm++ applications. *Projections' CPU usage profile* presents information about the overall workloads across processors. For each core, the average processing time (in percentage of time, green bar) is provided, as well as the global average (leftmost bar). In such graphics, the higher is the green bar, the higher is the CPU consumption for the corresponding core. For the Stencil3D application which is tightly coupled, this can lead to two interpretations. If the CPU consumption is almost homogeneous for the whole set of cores, this means that there are no cores that delay the other cores. Thus, in this case, the total processing time should be small and the average CPU usage should be high. On the opposite, if we observe that a set of cores has a higher CPU usage than the others, this means that these cores process the computation slower than the others. There is an imbalance in the capacity to compute and thus the average CPU usage is lower and total time is longer. Objects' migration time is not accounted into the processing time and thus accounts for a loss of computation time.

In the following figures we present the Projections visualization for the different variants of the platform. To improve readability, we do not present the results for the 256 cores but only the 8 first ones. However, this presentation is representative of what happens in the whole set of nodes. Figures 3 and 4 present the core utilization averages for *LBOff* and *RefineLB* on a homogeneous platform. Figures 5 and 6 correspond to the same experiment on a heterogeneous platform (half of the nodes has a lower CPU clock speed). Finally, Figures 7 and 8 are for the case where 1/8 of platform nodes are downclocked periodically.

With a homogeneous platform (Figures 3 and 4), there is little variation in CPU usage of the different *vnodes*, and *RefineLB* shows a higher processing time but a very stable CPU utilization. In that case, using a load balancer is not interesting and leads to a 3% lower average CPU utilization. With a static heterogeneous platform (Figures 5 and 6), the effects of load balancing are well visible. The CPU utilization is smoothed with *RefineLB* and the average CPU utilization of all *vnodes* is finally higher (12% improvement) than with no LB and total processing time is shorter. In the last experiment (Figures 7 and 8), the *vnode* 0 is downclocked periodically. For the test without LB, it is visible that *vnode* 0 is globally slower than others and, it delays *vnodes* 1, 2 and 3 that show a lower CPU utilization than the others. As in previous situations, using a good load balancing technique evens the CPU utilizations and finally leads to a 8% average CPU usage improvement and a shorter run time.

This experiment demonstrated that Distem enables experimenters to easily simulate perturbations and heterogeneity of nodes in order to evaluate load balancing strategies.

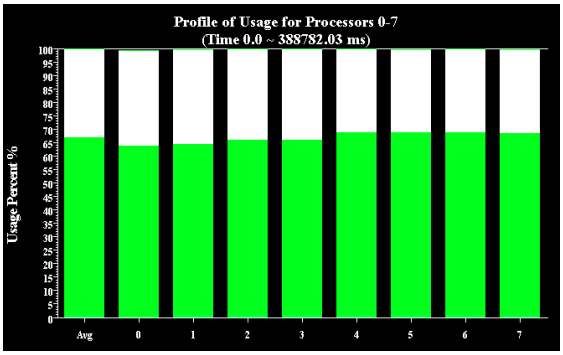


Figure 3: No load balancing on homogeneous platform. Total run time: 389 sec. Avg. CPU usage: 67%

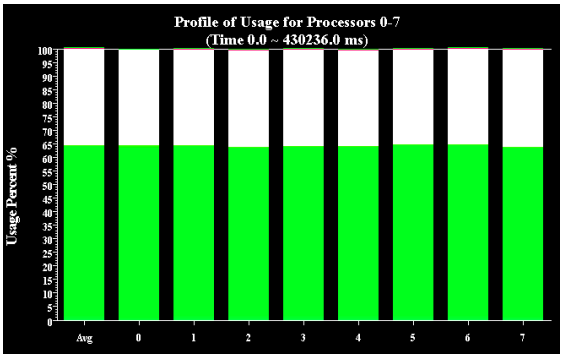


Figure 4: *RefineLB* on homogeneous platform. Total run time: 430 sec. Avg. CPU usage: 64%

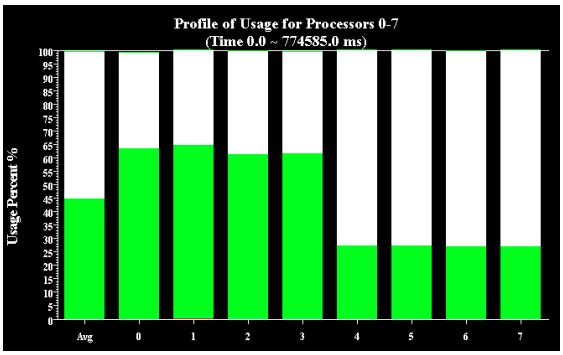


Figure 5: No load balancing with half nodes being Downclocked. Total run time: 775 sec. Avg. CPU usage: 45%

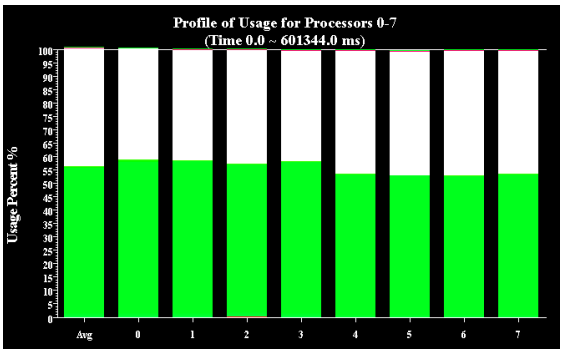


Figure 6: *RefineLB* with half nodes being Downclocked. Total run time: 601 sec. Avg. CPU usage: 57%

4 Related works

There exists a large number of works that aim at building experimental environments providing controlled performance of CPU or network. These are more extensively detailed in our previous works [2] and [17]. However, none of them provide the complete integration of the chain of tools for the full control of all the environment characteristics as Distem does.

In the context of HPC runtime evaluation, no standard method is used to run experiments. Below we survey some recently published papers by the Charm++ team to see how their experiments have been carried out.

[11] proposes to evaluate a distributed load balancer for Charm++ using an ad-hoc simulator and a real application trace from IBM BG/Q. Building one's own simulator and extracting trace from real application is time consuming. Simulation is valuable to deal with extreme scale platform, but it might not represent all the details of real applications.

[16] describes a temperature-aware load balancer for Charm++. Their experiments leverage the DVFS capabilities of processors to study the load balancer behavior. DVFS is not always available or is available only for few frequencies; the same study could have been carried out

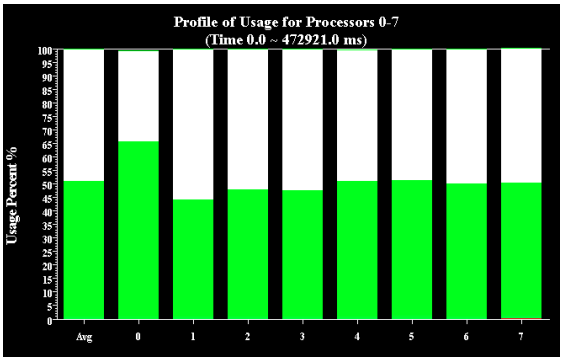


Figure 7: CPU Power Variation, no load balancing. Total run time: 473 sec. Avg. CPU usage: 51%

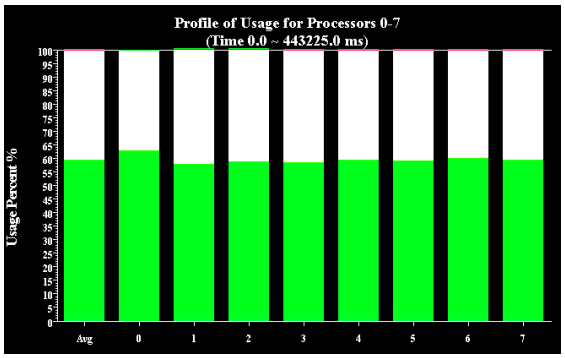


Figure 8: CPU Power Variation, *RefineLB*. Total run time: 443 sec. Avg. CPU usage: 59%

with Distem without specific hardware support. Indeed, Distem is able to emulate DVFS at core level. A similar experiment using DVFS is presented in [10]. However, it could not have been performed at the core level since DVFS was only available at the socket level. Such a limitation does not exist on a Distem platform.

[18] proposes to study network link utilization of different applications according to the underlying network topology. To create new network topologies, this work require advanced network management capabilities to switch off some links. Such experiment could be performed on any platform with Distem since virtual network topologies can be redefined.

[13, 14] present fault tolerance mechanisms on top of Charm++. Studying such properties of the HPC runtimes is probably one of the most difficult things. Generating realistic failures is not easy: one possibility is to mimic a failure in the application; another is to kill the application's processes, or to shutdown some nodes. Distem removes the burden of managing this manually by providing users with an event injection framework. This framework allows the experimenter to define realistic failure conditions, like MTBF, frequently used as a failure model in the literature.

5 Conclusions and future works

In this paper, we presented how Distem was extended to support the evaluation of HPC runtimes and applications. After detailing some relevant design parts of Distem with respect to the Exascale context (support for large scale experiments, and for evolving experimental conditions), we carried out several experiments, studying some key properties of HPC runtimes that are included on the targeted challenges to reach before 2020: (1) scalability of the OpenMPI runtime, (2) fault-tolerance and (3) load-balancing properties of Charm++ runtime. A lot of other experiments could have been carried out but the goal was to prove the usefulness of Distem with respect to enabling those experiments. In each experiment, we were able to draw conclusions that might be useful for the designers of runtimes. For instance, new load balancing strategies could be implemented in Charm++ in order to fit the next-generation HPC platform characteristics. Being able to execute experiments on a large set of platform configurations in a repeatable way, as made possible by Distem, is a strong basis to design and improve the HPC runtimes of the future.

Future works will follow three directions. First, we will work on conducting experiments with

100,000 and more nodes. Second, we would like to emulate more node characteristics, such as power consumption, heat dissipation or I/O performance, in order to perform realistic experiment that focus on future HPC and Cloud challenges. Finally, we look forward to working with HPC runtime and application developers in order to contribute to designing a better next generation of HPC software.

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on a Million Processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30. Springer, September 2009.

[2] T. Buchert, L. Nussbaum, and J. Gustedt. Methods for Emulation of Multi-core CPU Performance. In *HPCC-2011*, pages 288–295, September 2011.

[3] F. Cappello, E. Caron, M. Dayde, F. Desprez, et al. Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed. In *Grid'2005*, pages 99–106, November 2005.

[4] J. Dongarra, P. Beckman, T. Moore, et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computer Applications*, 25(1):3–60, February 2011.

[5] O. S. Gómez, N. Juristo, and S. Vegas. Replications Types in Experimental Disciplines. In *ESEM'10*, pages 3:1–3:10, October 2010.

[6] A. Gupta, O. Sarood, L. Kale, and D. Milojicic. Improving HPC Application Performance in Cloud through Dynamic Load Balancing. In *CCGRID'2013*, pages 402–409, May 2013.

[7] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA'93*, pages 91–108, September 1993.

[8] L. V. Kale and A. Sinha. Projections: A scalable performance tool. *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, April 1993.

[9] I. Koren and C. M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[10] H. Menon, B. Acun, S. De Gonzalo, O. Sarood, and L. Kale. Thermal aware automated load balancing for HPC applications. In *Cluster'2013*, pages 1–8, September 2013.

[11] H. Menon and L. V. Kale. A Distributed Dynamic Load Balancer for Iterative Applications. In *SC'2013*, pages 15:1–15:11, Denver, CO, USA, November 2013.

[12] R. Minnich and D. Rudish. Ten Million and One Penguins, or, Lessons Learned from booting millions of virtual machines on HPC systems. In *HPCVirt'2009*, March 2009.

[13] X. Ni, E. Meneses, N. Jain, and L. V. Kale. ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection. In *SC'2013*, pages 7:1–7:12, November 2013.

[14] X. Ni, E. Meneses, and L. V. Kale. Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm. In *Cluster'2012*, pages 364–372, September 2012.

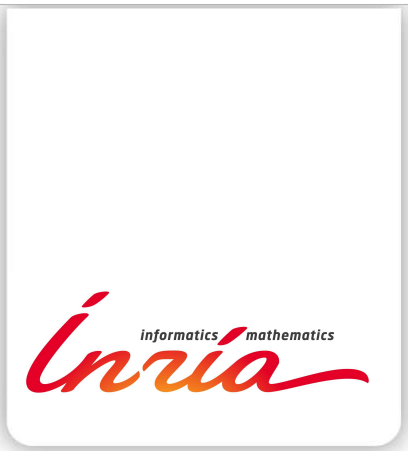
[15] Y. Robert. Models for fault-tolerance at very large scale, 2013.

[16] O. Sarood, E. Meneses, and L. V. Kale. A 'Cool' Way of Improving the Reliability of HPC Machines. In *SC'2013*, pages 58:1–58:12, Denver, CO, USA, November 2013.

[17] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In *PDP2013*, pages 172 – 179, Belfast, Royaume-Uni, February 2013.

[18] E. Totoni, N. Jain, and L. V. Kale. Toward Runtime Power Management of Exascale Networks by On/Off Control of Links. In *HPPAC'2013*, pages 915–922, May 2013.

[19] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *Cluster'2004*, pages 93–103, San Diego, CA, September 2004.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399