Distem: Evaluation of Fault Tolerance and Load Balancing Strategies in Real HPC Runtimes through Emulation

Cristian Ruiz, Joseph Emeras, Emmanuel Jeanvoine, Lucas Nussbaum

Distem: Evaluation of Fault Tolerance and Load Balancing Strategies in Real HPC Runtimes through Emulation

Cristian Ruiz, Joseph Emeras, Emmanuel Jeanvoine and Lucas Nussbaum

Inria, Villers-lès-Nancy, F-54600, France
Université de Lorraine, LORIA, F-54500, France
CNRS, LORIA - UMR 7503, F-54500, France
Email: firstname.lastname@inria.fr

***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 set of extensions to the Distem emulator that enable the evaluation of fault tolerance and load balancing mechanisms in such runtimes. Extensive experimentation showing the benefits of our approach has been performed with three HPC runtimes: Charm++, MPICH, and OpenMPI.**

Keywords*—*Experimentation, HPC runtimes, Fault tolerance, Load balancing, Emulation

I. 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 [1] 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.

Experimentation is fundamental in the HPC context. Evaluating current HPC runtimes under complex realistic conditions is critical for application developers and researchers.

This work contributes to enabling experimental evaluation of 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) introduction of heterogeneity and dynamic imbalance among the computing resources; (2) introduction of failures. Those features provide runtime designers with the ability to experiment their prototypes under a large range of conditions.

This paper is organized as follows. First, Section II provides an overview of Distem's design, and describes how it was extended to address the specific challenges of the evaluation of HPC runtimes. Then, Section III presents four experiments that demonstrate the benefit of such a tool in HPC runtime evaluation. Finally, Section IV compares our approach to other related works, and Section V concludes this paper and discusses our future work.

II. Design of an emulator to study HPC runtimes

This section first highlights the important points when studying HPC runtimes and presents the difficulties when evaluating them. Then, it present the design of Distem and how it enables testing HPC runtimes in heterogeneous environment, possibly under evolving conditions and in presence of failures.

A. Study of HPC runtimes

To design interesting features for studying HPC runtimes, we first have to delve what is important to achieve such studies and why it is difficult to achieve.

Compared to typical computing environments, the main characteristic of HPC environments is the high number of computing resources and its critical density. Furthermore, typical HPC workloads imply a large amount of computations that are likely to be executed for several days or months. Because of the high number of resources, the probability that failures hit one or more resources while computing is very high. High density of the resources also increases failures since it is complicated to extract the generated heat from computers and since heat is known to damage electronic components. Thus, it is quite obvious that HPC runtimes have to deal with failures.

To cope with the heat, hardware manufacturers and OS developers offer several solution that usually aim at reducing the computing power dynamically (for instance by reducing the frequency of some CPU cores, or by disabling some cores). For classical applications this has almost no impact but parallel codes usually used in HPC applications would be disrupted.

Indeed, a typical parallel application runs the same workload on all the computing units using network communications to synchronize and exchange required data among the computing units. To avoid inactivity of some computing CPU while other are performing a computation with a reduced computing power, HPC runtimes also have to implement load-balancing strategies.

In this paper, we will focus on the way to study fault-tolerance and load-balancing features in HPC runtimes.

When implementing such features, HPC runtime designers and developers tackle the difficulty to ensure that runtimes are working as expected and to tune them according to the execution platform.

Regarding fault-tolerance, fault injectors can be used to validate the recovery mechanisms. Those injectors can be embedded in the runtime but this often suffer from a lack of realism, like the one implemented in Charm++ [2]. Indeed, the runtime can stop some running processes but it cannot simulate an hardware or a network failure since it does not control the kernel. Some frameworks exist with various level of realism. However, realistic ones imply intrusive kernel modifications [3].

Regarding load-balancing, ad-hoc mechanisms are usually used, leveraging CPU scaling to reduce the frequency of some CPU. This approach lacks accuracy as it is hard to perform such operations on a lot of nodes at the required time to show precisely the benefit of a given load-balancing strategy. Furthermore, CPU scaling cannot set any frequency on any processor: only few frequencies are usually available, which can be an issue to subtly study a load-balancing strategy.

B. A framework for the evaluation of HPC applications

1) General overview of Distem: The features, general architecture and evaluation of Distem were presented in [4], 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 can either use CPU scaling to achieve that, or also a specific method designed to work independently from the native supported frequencies of a CPU. This is useful to evaluate precisely load-balancing strategies of HPC runtimes.

Leveraging IO throttling features of Linux, Distem is able to emulate slower IO. This could be interesting to evaluate checkpoint-based fault-tolerance strategies. We can imagine finding the best trade-off between taking checkpoints frequently (depending on the IO efficiency) and risk to restart an application from an older state.

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) Emulate heterogeneous and evolving experimental conditions: We have previously shown [4] that Distem can be used to create virtual heterogeneous platforms from a homogeneous cluster, with *vnodes* with different frequencies and number of cores, different IO capabilities and network links with different bandwidth and latency.

In the context of this work, we extended Distem so that those characteristics of the *vnodes* could be updated dynamically without redeploying the virtual platform. This is useful to achieve complex experiments where the platform is modified, like it could happen in the reality. Thus, evaluating CPU- or communication-aware load-balancers of HPC runtimes is possible and easy.

We have shown previously that Distem is based on lightweight virtualization. Leveraging Linux containers is actually a strong advantage since applications are executed on the same hardware as classical execution. This is not really the case when running applications on the top of classical virtualization solutions. Even if it tends to be improved, the architecture of the computer is virtualized in a complex way, which could induce a modified behavior for application executions.

3) Support for fault-tolerance: Handling failures is crucial for HPC runtimes, thus being able to generate realistic failures is required to perform accurate experiments.

Unexpected loss of nodes in a computing infrastructure is normally due to a hardware failure. Nodes can be lost in three different ways:

- **Graceful**: the node is shut down cleanly, using an operating system command. All application are notified and they have some graceful period to perform some cleaning tasks (usually some seconds).
- **Soft**: The node is forced to shut down. Applications running on the node as in the previous case are notified, however the node is shut down immediately. The application do not have the possibility to react.
- **Hard**: The node failed abruptly. The operating system is not able to terminate cleanly. The peer application are not notified (network communications are still seen as active until the failure is detected by the peer, using a timeout).

Distem is able to emulate these three behaviors using Linux containers and Netfilter. More specifically, the failures are implemented as follows:

- **Graceful**: classical stop of LXC containers (`lxc-stop`), this sends a `TERM` signal to all the processes belonging to the container. If applications are able to handle specifically this signal, they can perform some cleaning operations.

- **Soft**: forced stop a LXC containers (`lxc-stop -k`), this sends a `KILL` signal to all the processes belonging to the container. The effect is immediate.
- **Hard**: we drop all the packets received and sent by a *vnode* using `iptables`.

Speaking in the fault-tolerance terminology, **Graceful** and **Soft** failures are fail-stop failures, **Hard** failures are a kind of omission failures. Other kind of failures can be emulated, like performance failures that can be emulated by setting a very high latency on some virtual network interface or by radically decreasing the CPU performance of some nodes, implying very slow responses to requests. Distem is not able yet to generate byzantine failures but it could be added, if you consider the network point of view, leveraging Netfilter. Indeed, `libnetfilter_queue` offers an API to alter packets received by the kernel and allows to re-inject them. However one could argue that byzantine failures are not that likely in HPC context, thanks to the use of ECC memory that limits memory corruption, to well-tested software stacks, and to the use of infrastructures isolated from the Internet, limiting the risks of intrusion and malware.

4) Event injection framework: 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 (in the three different presented ways).

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.

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.

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.

III. Experimental evaluation

This section validates the benefits of Distem in the context of HPC runtimes evaluation. For that purpose we performed four kinds of experiments. First, we show how three HPC runtimes detect failures and behave when a process of the application is hit by a failure (Section III-A). Second, we validate the fault injection mechanisms of Distem depending on the degree of over-subscription (Section III-B). Third, we evaluate the fault-tolerance strategies implemented in Charm++ using a predetermined failure trace and probabilistic failures following several MTB values (Section III-C). Finally, we evaluate the behavior of some Charm++ load-balancers using Distem to create heterogeneous platforms and to dynamically change performance of nodes (Section III-D).

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 [1].

Experimental Setup: The experiments have been performed on three clusters of the Grid'5000 testbed [6]:

- *Graphene*, located in Nancy, composed of 144 nodes (1 Intel X3440 CPU at 2.53 GHz, 4 cores/CPU, 16GB of RAM, Gigabit Ethernet adapter);
- *Griffon* located in Nancy, composed of 92 nodes (2 Intel Xeon L5420 CPU at 2.5GHz, 4 cores/CPU, 16 GB of RAM, Gigabit Ethernet adapter);
- *Paravance* located in Rennes, composed of 72 nodes (2 Intel Xeon E5-2630v3 CPU at 2.4GHz, 8 cores/CPU, 128 GB of RAM, 10 Gigabit Ethernet adapter).

We used the following software stack: Debian Jessie, Linux kernel version 3.16, Charm++ version 6.6.0, OpenMPI 1.8.5 and MPICH 3.1. We used two applications for Charm++: a) *Jacobi3D* a 7-point 3-dimensional stencil that computes the transmission of heat on a three dimensional space. It implements in-memory checkpoint-restart [7] and b) *Stencil3D* a 6-point 3-dimensional stencil that iteratively average values in a 3D grid. This application has been chosen since it is relevant to observe the benefit of Charm's LB functionality. For the MPI based runtimes, we used the set of NAS benchmarks.

A. Evaluating failure detection of HPC runtimes

Failure	Framework					
	Charm++		OpenMPI		MPICH	
	Detected	Action	Detected	Action	Detected	Action
Graceful	Yes	C	Yes	H	Yes	E
Soft	Yes	C	Yes	H	Yes	E
Hard	No	-	Yes	H	Yes	E

TABLE I: Failure detection. C refers to the roll-back of the application to the previous checkpoint, H refers to the fact that processes hang, E refers to the termination of MPI processes

In this section, we use Distem to assess the failure detection capacity of HPC runtimes. The experiment consist in injecting a failure with Distem and observe if the failure is detected and what is the action taken by the runtime. We use *Jacobi3D* application for Charm++ and the NAS benchmarks for the MPI based runtimes. The parameters and specific application are not mentioned as the results remained unchanged for a large set of

[1] https://gforge.inria.fr/frs/?group_id=3519

```
cl.event_trace_add({ 'vnodename' => "node-1", 'type' => 'vnode' },'churn', { 80 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-5", 'type' => 'vnode' },'churn', { 120 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-9", 'type' => 'vnode' },'churn', { 130 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-13", 'type' => 'vnode' },'churn', { 140 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-17", 'type' => 'vnode' },'churn', { 150 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-21", 'type' => 'vnode' },'churn', { 160 => 'down' })
cl.event_trace_add({ 'vnodename' => "node-25", 'type' => 'vnode' },'churn', { 170 => 'down' })


begin
  cl.event_manager_start
rescue Distem::Lib::ClientError => e
  puts("Unable to start event manager (maybe it is already started?)")
end

cl.vnode_execute("node-1","time ./charmrun ++p #{num_procs} ++nodelist ~/nodelist ./jacobi3d 512 512 512 128 128 128")

cl.event_manager_stop
```

Listing 1: Framework for event injection. This listing shows an example where the Distem ability to inject events is used in the evaluation a Charm++ application.

applications and parameters. In the following part, we discuss our findings.

Charm++ detects *Graceful* and *Soft* failures. Charm++ reacts in the same way to both kind of failures: it rolls back the execution to the previous checkpoint. In the first case the runtime catches the signal but it ignores it. When *Hard* failures are injected Charm++ is unable to detect them and the application freezes [2].

OpenMPI detects the three types of failures. However, the runtime does not abort the execution. For the first two types of failures the runtime clean up all the processes running on the different nodes but the main application blocks. In the presence of a *Hard* failure the runtimes does not clean the processes running on the other machines [3].

MPICH detects the three types of failures. It aborts the execution when a failure is detected which is the expected behavior.

The results are summarized in Table I. *MPICH* is the only runtime that aborts its execution in the presence of *Hard* failures. We observe that three runtimes have the same behavior under the presence of *Graceful* and *Soft* failures. In the case of Charm++ runtime, we looked at the execution traces and found that the runtime has around 10 seconds to react. A checkpoint could be done given that this operation takes normally less than one second as well as a respective migration operation in order to avoid losing any replica and to improve the resilience. Fault tolerance of MPI based runtimes is still not widely available, therefore aborting the execution is the best option that could be taken. Production clusters could see their utilization reduced since jobs will remained blocked if the runtime does not abort. Thus the failure will not only affect the current user but it will prevent the rest of the users to use the remaining resources.

Testing HPC runtimes under the presence of failures is very important and it has to be integrated into the software development cycle. We foresee Distem as a way to alleviate this task. Distem simplified the uncovering of problems in the failure handling for widely used HPC runtimes.

[2]Charm++ developer team was notified about this behavior

[3]This problem has been fixed in the latest OpenMPI version 1.10.1

B. *Validity of fault injection mechanism*

In this experiment, we validate the Distem fault injection mechanism.

To do so, we compare three mechanisms of fault injection:

- the mechanism built into Charm++;
- real faults using a classical OS reboot (`/sbin/reboot`);
- Distem fault injection mechanism when running 1,2, or 4 containers per physical machine.

We run 1000 iterations of *Jacobi3D* using 64 physical machines in the *Graphene* cluster. The checkpoint frequency was set up to each 100 iterations. A failure trace was defined, that injects faults after the first, second and third checkpoint. Faults here, refer to **Soft** shutdown. The Charm++ runtime ensures that Charm++ objects on a node are replicated in memory on some other nodes at each checkpoint. We inject a total of 10 failures that were distributed in a way that the probability of making replica nodes crash is reduced. The same trace of failures is used for the three mechanisms (see Table II).

Node killed	Time (secs)
node-1	80
node-5	120
node-9	130
node-13	140
node-17	150
node-21	160
node-25	170
node-29	220
node-33	230
node-37	280

TABLE II: Trace of the events injected while evaluating the Charm++ application in Section III-B

Mechanism	% termination	Mean walltime (secs)
Charm++ Injection	100%	268.55
Real Injection	66%	267.19
Distem 1vn	56%	286.43
Distem 2vn	50%	287.05
Distem 4vn	56%	294.45

TABLE III: Percentage of successful application executions

In Table III we observe that Charm++ fault injection provides a very optimistic and unrealistic scenario where

the application always finishes its execution successfully. We include in the table the walltime of the application in order to show the overhead introduced by Distem which goes from 6.17% to 9.25% due to the use of containers and the degree of over-subscription. Additionally, it is shown that the walltime remains in an acceptable range to be meaningful and to be comparable with real injection. In this experiment, we show that fault injection mechanisms at the application level have to be complemented with a more realistic approach.

We demonstrate that Distem offers realistic experimental conditions and that the level of realism is maintained in the presence of over-subscription. Listing 1 shows a snippet of the script used for this experiment. We can observe the API provided by Distem which helps to define this scenario programmatically.

C. Study of fault tolerance strategies in Charm++

In this experiment, we use Distem to evaluate the fault tolerance capacity of the Charm++ runtime [2]. The experiment has been divided in two parts. The first part uses a static and predetermined trace of failures like described in the previous section whereas the second part uses events that are generated following a probabilistic law. We use the event generator described in Section II-B2 which is able to manage these two aforementioned scenarios.

1) Trace of predetermined failures: For this experiment, we use the *Graphene* cluster. We run 1000 iterations of *Jacobi3D* on 128 cores (32 nodes, 4 *vnodes* per node, 1 core per *vnode*) with a problem size of $1024 \times 1024 \times 1024$ and a decomposition of 128^3 and configured with different frequencies for the checkpoint: $\{10, 25, 50, 100, 120, 130, 140\}$ iterations. Failures are injected following the events presented in Table IV.

This experiment evaluates the cost and trade-off of different checkpoint frequencies. This is summarized in the resulting application walltime (Figure 1). The most important overheads are at both extremes of the range: 10 and 140 which represent the overhead of performing a checkpoint very often and the overhead of loosing a large number of iterations. The overhead due to the loss of iterations is around 20% larger than the loss due to checkpointing.

This experiment illustrates how Distem can be used to fully automate the tunning of fault tolerant applications by choosing the best checkpointing frequency. This is necessary given that the cost of checkpointing will vary according to the application.

Node killed	Time (secs)
node-1	80
node-21	160
node-41	240
node-61	320
node-81	400
node-101	480

TABLE IV: Trace of the events injected while evaluating the Charm++ application in Section III-C

2) Trace of failures that follow MTBF: For this experiment, we use the *Griffon* cluster. 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 set the frequency of checkpointing to 200 iterations.

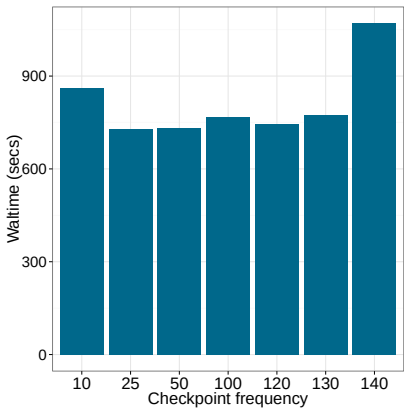


Fig. 1: Overhead of different checkpoint frequencies under the presence of failures

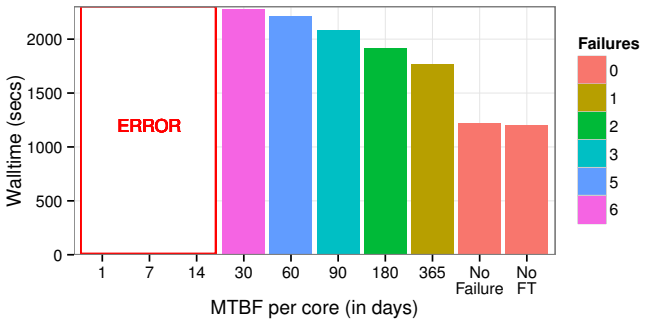


Fig. 2: Execution runtime of *Jacobi3D* application under different values of MTBF.

In this experment instead of using a deterministic trace we use a probabilistic distribution: the Weibull distribution. This distribution is suitable for approximating MTBF behaviors [8], [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 (as already shown in the previous experiment). The replication only induces a slowdown of a few seconds on a 20 minutes run.

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 V, which presents the number

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 V: Number of iterations lost in failures.

of iterations lost at a given failure[4], 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. All values in Table V are examples of the iterations lost due to the failures.

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. Using probabilistic law to generate events is suitable to achieve realistic evaluations. However, depending on the time and the scale of an experiment, results might not be fully reproducible because of the probabilistic aspect.

D. 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 used 8 physical machines from *Paravance* cluster. For each machine a vnode is configured per core, making a total of 128 vnodes. We run 100 iterations of *Stencil3D* application with a 1024^3 problem size and a 128^3 decomposition. We create two different scenarios:

- *heterogeneous*: A platform where half of the *vnodes* have a CPU clock reduced by Distem from 2.4GHz to 1.2GHz. This case is an interesting study of what would be the result of running Stencil3D on 2 cluster partitions with different CPU clock speeds.
- *dynamic*: A homogeneous platform where the available CPU power of a sub-part 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 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 [10].

For both scenarios we evaluate two load balancers (*RefineLB* and *Hybrid*) provided by Charm++. Therefore we test the following variations:

[4]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.

- **LBOff**: load balancing is deactivated. Charm++ objects will not be migrated and no processor imbalance check will be done by Charm++ runtime.
- **RefineLB**: centralized load balancer that moves objects away from the most overloaded processors to balance the workload. It also limits the number of objects being migrated.
- **Hybrid**: distributed load balancer that uses a hierarchical strategy.

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.

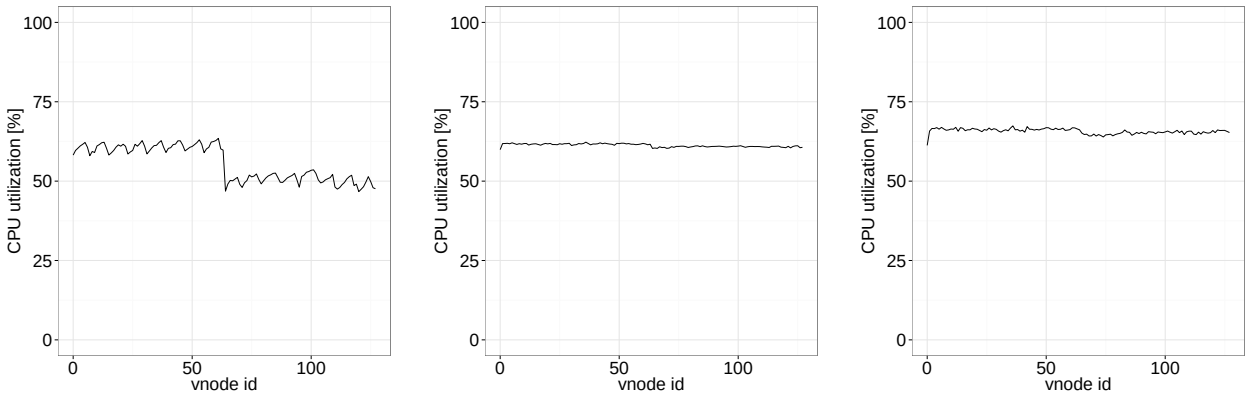
Figure 3 shows the CPU utilization for each of the 128 vnodes in the two scenarios and the different variations. Figure 3a shows the CPU utilization for an heterogeneous hardware with not load balancer activated (*LBOff*). Figures 3b,3c depict a smooth curve for the CPU utilization as a result of the activation of the load balancers. Although *Hybrid* shows a better CPU utilization, the final runtime of the application is bigger than the normal execution time (*LBOff*). This is due to CPU utilization generated by the load balancer itself. *RefineLB* shows a CPU utilization of 5.67% and a reduction of 6.40% in the walltime. Figure 3d shows the CPU utilization for a *dynamic* platform. Again the curve of utilization is smoothed by both load balancers. We observe here, as in the previous case that *RefineLB* behaves better than *Hybrid*. In this experiment, 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. Finally, the difference in term of walltime for both load balancers is shown in Figure 4.

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

IV. 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 [11] and [4]. 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.

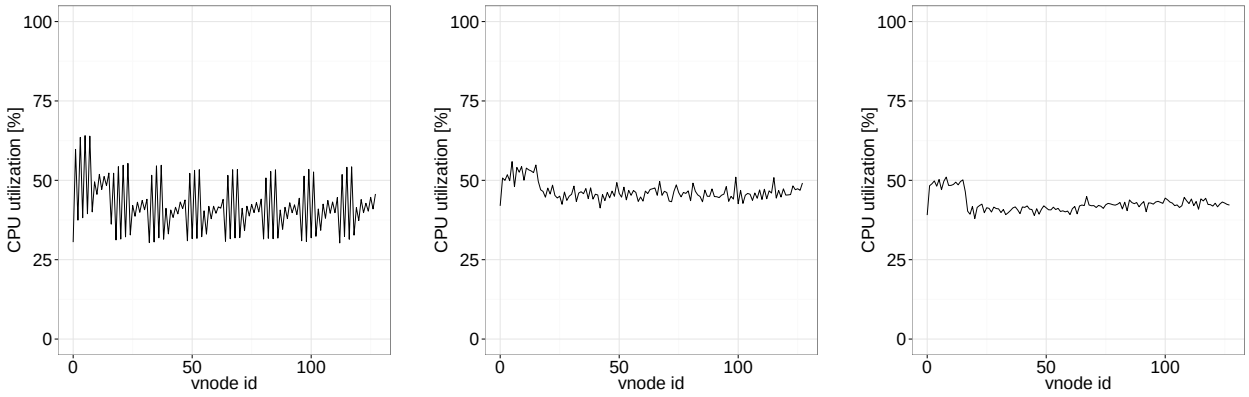
In the context of HPC runtime evaluation, no standard method is used to run experiments. As the Charm++ team is seen by most as doing state-of-the-art research in the field of HPC runtimes, we use them as an example and survey some of their recently published papers to see how their experiments have been carried out.



(a) LBOff - Walltime: 341.51 secs
Average utilization: 55.58%

(b) RefineLB - Walltime: 320.96 secs
Average utilization: 61.25%

(c) Hybrid - Walltime: 356.04 secs
Average utilization: 65.71%



(d) LBOff - Walltime: 347.52 secs
Average utilization: 41.87%

(e) RefineLB - Walltime: 322.67 secs
Average utilization: 46.55%

(f) Hybrid - Walltime: 359.07 secs
Average utilization: 42.65%

Fig. 3: Figures (a), (b) and (c) represent the execution in a virtual infrastructure where half of the nodes CPU clock is reduced to half of the frequency. Figures (d), (e) and (f) represent the execution in a virtual infrastructure where the CPU clock of some machines is varying.

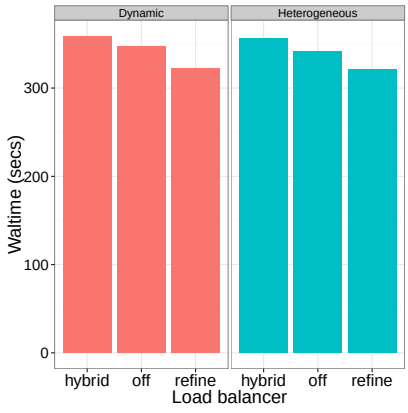


Fig. 4: Walltime comparison of using *RefineLB* and *Hybrid* load balancers in a *Dynamic* and *Heterogeneous* environments

[12] 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.

[13] 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 with Distem without specific hardware support. Indeed, Distem is able to emulate DVFS at core level. A similar experiment using DVFS is presented in [14]. 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.

[15] 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.

[16] and [17] 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.

V. 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 HPC context (heterogeneous and evolving experimental conditions, realistic fault-injections), we carried out several experiments, studying two key properties of HPC runtimes that are included on the targeted challenges to reach before 2020: (1) fault-tolerance and (2) load-balancing properties of HPC runtimes. 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 sound basis to design and improve the HPC runtimes of the future.

Future works will follow three directions. First, we will carry out experiments related to I/O evaluation. As seen in the paper, it would be an interesting feature to evaluate mechanisms using on-disk checkpointing. Distem already has support for I/O throttling that leverages a Linux feature called `blkio` but extensive evaluations still have to be achieved and maybe some design refactoring will have to be performed since in some cases the throttling might not work as expected (case of buffered I/O for instance). Second, we would like to emulate more node characteristics, such as power consumption, heat dissipation in order to perform realistic experiment that focus on future HPC and Cloud challenges like power-aware scheduling. We could also imagine introducing an energy cost model to be able to evaluate fancy scheduling/load-balancing algorithms like *follow-the-sun* algorithms. 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] J. Dongarra, P. Beckman, T. Moore *et al.*, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computer Applications*, vol. 25, no. 1, pp. 3–60, February 2011.

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

[3] C. Coti and N. Grenèche, "Os-level failure injection with systemtap," *CoRR*, vol. abs/1502.01509, 2015. [Online]. Available: http://arxiv.org/abs/1502.01509

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

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

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

[7] 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*, San Diego, CA, September 2004, pp. 93–103.

[8] Y. Robert, "Models for fault-tolerance at very large scale," University of Tennessee Knoxville, 2013.

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

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

[11] T. Buchert, L. Nussbaum, and J. Gustedt, "Methods for Emulation of Multi-core CPU Performance," in *HPCC-2011*, September 2011, pp. 288–295.

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

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

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

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

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

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