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

Cristian Ruiz, Joseph Emeras, Emmanuel Jeanvoine, Lucas Nussbaum

To cite this version:


HAL Id: hal-00949762
https://inria.hal.science/hal-00949762v3
Submitted on 6 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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

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 describes how the Distem emulator was extended to address the specific challenges of the evaluation of HPC runtimes. Then, Section III presents experiments that demonstrate the benefit of such a tool for HPC runtime evaluation. Finally, Section IV compares our approach to 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 presents the design of Distem and how it enables testing HPC runtimes in heterogeneous environments, 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 solutions 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.

### 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 failed abruptly. The operating system is notified (network communications are still seen as active until the failure is detected by the peer, using a timeout).
- **Soft**: 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).
- **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 of 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. Performance failures can be emulated as well 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.

### 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.

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[2]:

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

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 [6] 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

<table>
<thead>
<tr>
<th>Failure</th>
<th>Charm++</th>
<th>OpenMPI</th>
<th>MPICH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graceful</td>
<td>Detected</td>
<td>Action</td>
<td>Detected</td>
</tr>
<tr>
<td>Soft</td>
<td>Yes</td>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>Hard</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 applications and parameters.

The results are summarized in Table 1. 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. 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. MPICH detects the three types of failures. It aborts the execution when a failure is detected which is the expected behavior.

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

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: a) the mechanism built into Charm++, b) real faults using a classical OS reboot (/sbin/reboot), c) 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. 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.

1https://gforge.inria.fr/frs/?group_id=3519
2Experiments presented in this paper were carried out using the Grid’s 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).

3Charm++ developer team was notified about this behavior
4This problem has been fixed in the latest OpenMPI version 1.10.1
In Table I, 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.

### 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.

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 tuning 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.

2) Trace of failures that follow MTBF: For this experiment, we use the Griffin 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.

In this experiment instead of using a deterministic trace we use a probabilistic distribution: the Weibull distribution. This distribution is suitable for approximating MTBF behaviors [7, 8]. 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.

###TABLE III: Number of iterations lost in failures.

<table>
<thead>
<tr>
<th>MTBF</th>
<th>Iterations Lost at Failure</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>365</td>
<td>40</td>
<td>140</td>
</tr>
<tr>
<td>180</td>
<td>140</td>
<td>110</td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>60</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>50</td>
<td>0</td>
</tr>
</tbody>
</table>

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 III, which presents the number of iterations lost at a given failure, 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. This is due to the fact that failure detection and restart preparation took between 10 and 20 seconds instead of the 3 seconds measured in the other cases.

In this experiment, thanks to Distem’s failure generation framework, we were able to easily simulate node failures

---

3 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.
with different MTBF values and thus to observe Charm’s FT efficiency. Using a 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 of the 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 in a data center or CPU performance variation due to oversubscription.

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

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). It is observed that half of the vnodes have a lower CPU utilization which is the result of running a tightly coupled application in an heterogeneous platform. Figures 3b and 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.

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 [10] 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.

[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.

[12] 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 [13]. However, it could not be performed at the core level since DVFS was only available at the socket level. Such a limitation does not exist on a Distem platform.

[14] and [15] 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 experimenters to define realistic failure conditions, like MTBF, frequently used as a failure model in the literature.

[16] proposes to study network link utilization of different applications according to the underlying network topology. To create new network topologies, this work required 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.

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 experiments 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.

REFERENCES


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.