The MIG Framework: Enabling Transparent Process Migration in Open MPI

Federico Reghenzani
federico1.reghenzani@mail.polimi.it

Gianmario Pozzi
gianmario.pozzi@mail.polimi.it

Giuseppe Massari
giuseppe.massari@polimi.it

Simone Libutti
simone.libutti@polimi.it

William Fornaciari
william.fornaciari@polimi.it

DEIB Politecnico di Milano
via Ponzio 34/5
Milano, Italy

ABSTRACT

This paper introduces the `mig` framework: an Open MPI extension to transparently support the migration of application processes, over different nodes of a distributed High-Performance Computing (HPC) system. The framework provides mechanism on top of which suitable resource managers can implement policies to react to hardware faults, address performance variability, improve resource utilization, perform a fine-grained load balancing and power thermal management.

Compared to other state-of-the-art approaches, the `mig` framework does not require changes in the application code. Moreover, it is highly maintainable, since it is mainly a self-contained solution that has required a very few changes in other already existing Open MPI frameworks. Experimental results have shown that the proposed extension does not introduce significant overhead in the application execution, while the penalty due to performing a migration can be properly taken into account by a resource manager.

CCS Concepts

•Software and its engineering → Checkpoint / restart; Multiprocessing / multiprogramming / multitasking; •Computing methodologies → Parallel computing methodologies; •Computer systems organization → *Distributed architectures;*

Keywords

Process Migration; Scheduling; Message Passing; Distributed Systems; Open MPI; MPI; HPC; Runtime

1. INTRODUCTION

Given the evolution of High-Performance Computing (HPC) systems and silicon technology, modern and future parallel systems must deal with two major concerns: the increasing number of computing nodes, thus CPU cores, and the end of Dennard's scaling [1], with the following issues related to power dissipation. These two aspects are posing new challenges in terms of performance scaling and effective nodes utilization, power and thermal management, reliability and fault-tolerance.

Concerning reliability and fault-tolerance, the Mean Time Between Failures (MTBF) of current supercomputing systems is already way below 100 hours [2, 3, 4]. This means that the availability of fault-tolerance techniques is particularly crucial to guarantee the safe execution of long-running workloads. In this regard, a very common solution is to exploit *Checkpoint/Restart (C/R)* based approaches, where the execution state of the application is periodically saved (checkpoint) so that, in case of fault, it can be resumed (restart) from the last consistent state.

An alternative to C/R is to migrate the execution of the application from a faulty to a more reliable set of nodes. This is a versatile technique since a system can benefit from task migration support, not only to react to faults, but also for resource management purposes, e.g. load balancing.

Furthermore, we must consider that the effective exploitation of an extremely parallel HPC system requires suitable programming models (e.g., MPI). These need to be properly supported by a run-time resource manager in charge of driving the task placement (scheduling and mapping) over the wide set of available computing resources. It turns out that, if the resource manager can rely on a migration mechanism, it can also dynamically change the computing nodes assignment during the execution of the applications. This opens up a wide range of possibilities. For instance we can, not only react to faults, but also adapt the computing resources assignment to time varying application performance requirements. Moreover, the application load can be balanced among the system nodes, in order to level down the power consumption and the temperature peaks. Ultimately, introducing task migration support in HPC systems enables the implementation of solutions to address all the aforementioned challenges.

In this paper we describe the `mig` framework, an Open MPI extension that introduces a novel process migration mechanism that is as transparent as possible with respect to applications and other Open MPI internals. The proposed framework implements a system-level migration schema based

EuroMPI '16, September 25-28, 2016, Edinburgh, United Kingdom

DOI: http://dx.doi.org/10.1145/2966884.2966903

on the idea of grouping the MPI application processes belonging to the same node into multiple migratable entities. The group size can be set at launch time by the run-time resource manager taking into account that, as shown in Section 4, the overhead can consequently vary. This schema allows us to perform fine-grained migrations, i.e., to migrate parts of a single application onto a different nodes, where the migrated processes execution is then resumed.

The paper is structured as follows: Section 2 reviews previous works proposing task/process migration solutions for Open MPI or HPC systems. Section 3 describes the `mig` framework along with the changes introduced in the Open MPI framework and needed to support the module. Section 4 reports the results of a set of experiments aiming at evaluating the overheads introduced by `mig`. Finally, Section 5 presents some final remarks and possible future developments.

2. RELATED WORKS

Several C/R approaches for parallel applications have already been proposed and implemented [3]. Most of the C/R implementations rely on *Berkley Lab's Checkpoint/Restart* kernel-space tool (BLCR) [5] and the *libckpt* user-space library [6].

C/R based approaches usually adopt the following schema: 1) synchronization of the application processes execution to reach a global consistent state; 2) application execution state saving (checkpoint); and 3) application execution resuming. In case of fault, all the running processes are killed and the application is restarted resuming the state from the last checkpoint.

C/R mechanisms can be managed either at *application* or *system-level*. In the former case, also known as user-level, the application itself is in charge of synchronizing the execution of its processes and performing the checkpoint. This is typically done by calling suitable library functions. In the latter case, instead, this task is accomplished by the run-time system that controls the application life-cycle, e.g. the resource manager or the run-time programming model support (e.g., MPI runtime).

Hursey *et al.* [7] extend the Open MPI stack with additional layers that provide C/R capabilities in a network agnostic fashion. The application processes can be stopped and then restarted on a different set of nodes potentially characterized by a different network topology. This solution introduces notable code dependencies between internal Open MPI frameworks. Moreover, it induces significant overheads: copying the process state images on an external storage server, becomes in fact a real bottleneck for the system. This drawback, along with the poor maintainability of the software, led the Open MPI developers to disable these additional layers since Open MPI version 1.7.

The common limitation of C/R based approaches is the overhead introduced by performing periodical checkpoint, which is done during the entire application lifespan. In some use cases this overhead impacts dramatically, even doubling the execution time of the applications [4]. A further problem is that this side-effect increases exponentially with the system size, i.e. the number of computing nodes. Considering a large HPC system with thousands of nodes and not negligible power supply costs, the overhead must be evaluated not only in terms of time, but also in terms of energy consumption [8].

For these reasons, an alternative approach to stop and resume the execution of parallel applications relies on *task migration* techniques. Such techniques can be classified on a granularity basis. Task migrations in fact can be performed either at *virtual-machine*, *container* or *process-level*. The first two classes are very common because executing MPI applications over virtual machines and containers simplifies the workload management, other than providing isolation guarantees. However virtual machines also have a significant impact on the application performance, especially for I/O intensive workloads[9]. The lack of shared memory communication between processes on different virtual machines indeed has been considered an important inefficiency since the beginning of virtualization's use in HPC environments[10]. Although many approaches have been proposed to mitigate this problem, the significant overhead persists even today, inducing an increment of latency by a factor up to 16x for communication intensive operations [11].

The third class, i.e. process-level migration, allows us to potentially achieve a higher utilization of the system resources, with respect to virtualization based approaches. Concerning this class, the most promising solution has been proposed by Wang *et al.* [2]. The basic idea of the authors is to try to minimize the number of C/R by using a proactive approach: health monitoring of the computing node state and migration of all the running processes on a different node, in case of imminent fault prediction. Their approach reduces the number of performed C/R with respect to periodical checkpoint based techniques. However this solution requires to synchronize all the running processes into a global consistent state, before stopping and migrating them to a new node; which can represents a problems in case of imminent faults. The approach was implemented in LAM/MPI (predecessor of Open MPI) using the BLCR tool.

According to previous works, we can argue that the main issues to be tackled when dealing with processes migration in HPC systems are:

- Design an easy to maintain migration framework;
- Enable migration support without introducing changes in the application code;
- Enable the possibility of migrating just part of the application, i.e. a subset of processes;
- Do not bound the migration overhead to the synchronization of the processes execution into a global consistent state;
- Provide interfaces to allow a resource manager to drive the processes migration according to smart policies.

The solution we propose addresses all the aforementioned issues: 1) it is a process-level migration mechanism whose granularity can be tuned by the resource manager; 2) it does not require any change to the application code; 3) the migration is almost completely transparent with respect to the application execution; and 4) migration can be triggered by a resource manager through a suitable API.

We implemented our proposed task migration approach as a not invasive Open MPI additional framework, that we called `mig`.

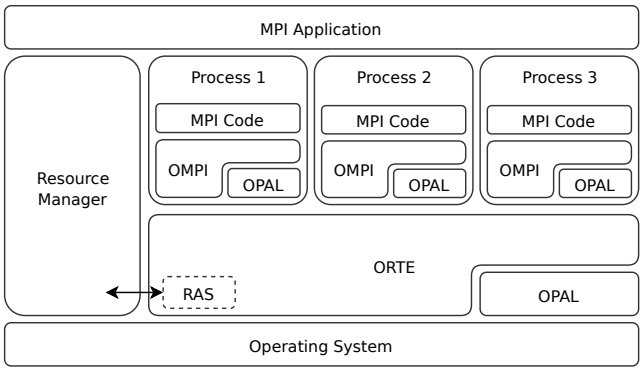

Figure 1: The architecture of Open MPI modules and interaction with the resource manager

3. DESIGN AND IMPLEMENTATION

In this work we present the Open MPI framework with a mean to perform process migration. What triggers migrations is not addressed, but we assume that a resource manager is in charge of determining when the migration of some processes is necessary, e.g. because a fault has been detected (or predicted) or because a rescheduling has been triggered.

The idea is that the resource manager could indeed signal a migration request to the Open MPI runtime by sending a `(source node, destination node)` pair via the already existent socket channel. Then, the Open MPI runtime, in particular the `mig` framework, performs the sequence of actions needed to actually migrate the application processes.

We use the *Checkpoint/Restore In Userspace (CRIU)* tool [12] in order to perform the checkpoint/restore of the execution of the processes. CRIU is a recent C/R tool, developed by the OpenVZ team, that is gaining a lot of interest in PaaS-level virtualization environments [13]. The big advantage of CRIU is that the required kernel-space changes have been already integrated in the Linux kernel. Its use therefore does not require any extra module: the checkpoint and the restore operations are completely performed in user-space, making it a portable solution. This, together with the very active developers community, motivated us to choose this tool for implementing the C/R functionalities in our migration framework.

3.1 Open MPI architecture

Open MPI is an MPI implementation based on a Modular Component Architecture (MCA) [14]: the main functional parts of Open MPI are divided in *frameworks*. Each framework contains one or more *components*. Depending on user selection and system capabilities, at run-time each component can be enabled or not.

The frameworks are grouped in three layers: *Open MPI (OMPI)*, that provides the application-level API over the runtime environment; *Open Run-Time Environment (ORTE)*, that is the underlying subsystem controlling the life-cycle of each application process; *OPAL*, that is an utility library. The overall architecture is shown in Figure 1.

The `mpirun` command is used to launch an MPI application. The node from which an application is started is called *Head Node Process* (HNP) and manages the entire program execution. When an Open MPI application is launched, `mpirun` starts the ORTE daemons – called `orted` – which is in charge of controlling the life-cycle of the processes to spawn on the local node. Instances of ORTE daemon can also be started on remote nodes. The set of available nodes is provided to ORTE through the *Resources Allocation Subsystem* (`ras`) framework. Communications between HNP and remote daemons go through a set of ORTE frameworks, respectively listed in descending order of network services they offer: `plm` (Process Lifetime Management), `rml` (Runtime Messaging Layer) and `oob` (Out Of Band).

Application-level messaging is managed by *OMPI* subsystems: `pml` (P2P Management Layer) catches MPI calls and forwards it to one of the active `btl` (Byte Transfer Layer) components – e.g., *TCP*, shared memory, *InfiniBand*. During the application startup each point-to-point channel is assigned to the fastest `btl` component depending on the availability.

3.2 Open MPI extension

At the time of writing, Open MPI is composed of more than 50 frameworks, each of them having between 1 to 10 components. For this reason we decided to add the migration support reducing as much as possible the impact on other frameworks. This choices is not only dictated by the need of reducing overheads, but also of limiting the introduction of hard-to-maintain code. As previously reported, Hursey's work was indeed rapidly removed from the mainline repository of Open MPI, because it was too demanding to maintain. Here we will detail the frameworks involved in our work:

- `ras` (part of ORTE): it provides the communication channel between the Open MPI runtime and the resource manager. Currently, Open MPI uses this framework only in the application initialization, to get the full list of available nodes. We extended the `ras` API to allow the resource manager to send migration requests during the applications execution and to be notified about the status of the requests.

- `mig` (part of ORTE): it is the framework we implemented to enable the migration mechanisms. The provided functionalities are controlled by `ras`on behalf of the resource manager. `mig` is in charge of coordinating the migration phases via `plm`, basically routing commands to the ORTE daemon instances involved. `mig` is also responsible of performing checkpoint/restore and of sending the process status image to the destination node. Just like other frameworks in Open MPI, `mig` is composed of a *base* component, containing the code implementing the common functionalities, and a specialized component for C/R based on CRIU. In this way, adding the support for a new C/R mechanism only requires the addition of a new component and does not introduce further changes in the framework core.

- `oob` (part of ORTE): the "out-of-band" framework provides the low-level API for the communication between HNP $\Leftrightarrow$ `orted`, and `orted` $\Leftrightarrow$ its child processes. The current Open MPI implementation includes TCP as the only transport layer for ORTE daemons inter- communication. This framework contributes to the process migration by managing the opening and closure of the

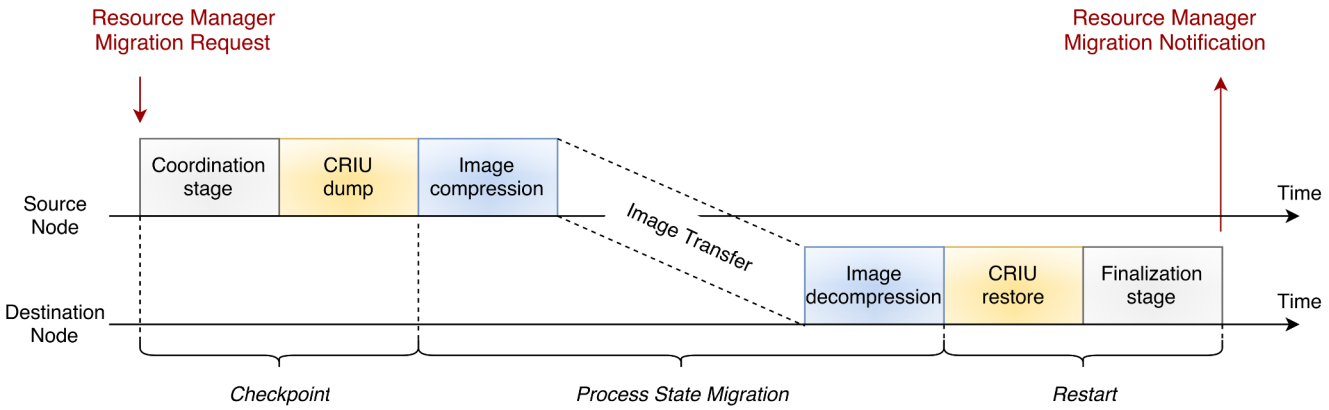

Figure 2: The migration phases

pending TCP socket connections towards the migrating ORTE daemon instance.

- `plm` (part of ORTE): high-level HNP ⇔ `orted` communication framework. We implemented the protocol necessary to coordinate the ORTE daemon instances in the *base* component. We also added the `ssh` call that spawns the `orted-restore` daemon on the destination node. This daemon is in charge of resuming the processes execution once the checkpoint image transfer is completed.
- `btl` (part of OMPI): this is the application-level peer-to-peer communication framework. In our work we modified the *TCP* component to manage the opening/closure of the TCP socket connections among migrating application processes.

Multiple ORTE daemon instances

The Open MPI default behaviour is to instantiate a single ORTE daemon (`orted`) for each allocated node of the HPC system, so that each `orted` manages all the processes running on its node. Our migration approach consists in moving an ORTE daemon and all the processes it is managing from the current node to another one. This means that the migration approach works at `orted`-level granularity. Therefore, at node level, grouping Open MPI processes of the same applications on top of multiple ORTE daemons allows us to migrate just a subset of the application processes. We modified the framework in order to let the resource manager specify how many daemon instances to start on each node, thus implicitly selecting the migration granularity.

The major overhead introduced by running multiple ORTE daemon instances on the same node is due to the lack of shared memory communication between processes managed by different daemon instances. However, if the resource manager does not require to split processes in multiple ORTE daemon instances then no extra overhead is introduced in the application execution.

3.3 CRIU

As argued above, the CRIU library is used in the corresponding C/R component of the `mig` framework to perform checkpoint/restart of a single ORTE daemon execution and its children, i.e. the Open MPI application processes.

The checkpoint stage – called *dump* in CRIU – freezes the processes execution and creates a collection of binary files containing the processes state. In this collection we can distinguish three categories of files: *inventory*, *image*, and *auxiliary*. CRIU stores the meta-data needed to perform the restore in *inventory* and *auxiliary* files. The *image* files are instead the memory dump of the processes and contain all the OS-level information, such as file descriptors, file-system mount-points, signal masks and ghost files[1].

The restart stage – called *restore* in CRIU – reads the binary files previously generated by the *dump* stage and restarts the frozen processes. This operation is not straightforward if the restore occurs on a node different from where the checkpoint has been performed: program executable, libraries and data files must be in fact present and identical in the destination node. Moreover, remote file-systems must be mounted and the process identification numbers (PIDs) must be available because the processes cannot change their PIDs after the restore. Given that, there is no guarantee about the fact that the PIDs of the migrating processes have not been already assigned on the destination node, we exploited the *Linux Namespaces*, a feature of the Linux Kernel [15] that allows us to isolate a set of processes in a detached environment via the `unshare` system call. In order to do this, the `orted-restore` executes the following C call:

```
unshare(CLONE_NEWNS | CLONE_NEWPID)
```

where the `CLONE_NEWNS` flag detaches the mount namespace and the `CLONE_NEWPID` flag detaches the PID namespace; hence, `orted-restore` and its children share the new private namespaces.

The `orted-restore` daemon becomes then the `init` process (PID=1) in the new empty PID namespace of the destination node and can restart the application processes with the original PIDs from the source node. The isolated mount namespace is necessary to remount the `/proc` directory in order to match the new process identifier configuration. At this point, the ORTE daemon instance and its children can be safely restarted.

[1]In Linux, files may not have a name, e.g., `unlink`-ed but still opened files, shared memory files and over-mounted paths.

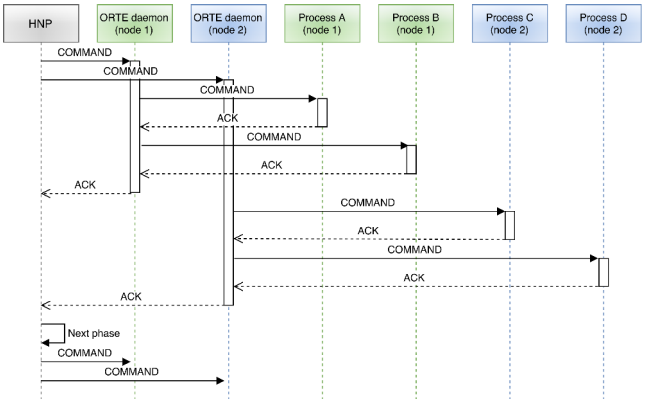

Figure 3: Sequence diagram migration messages exchange

3.4 Migration phases

In this section we describe in detail how the proposed migration mechanism is structured. We have divided the migration procedure in five phases: 1) *Coordination stage*; 2) *CRIU dump*; 3) *Process state migration*; 4) *CRIU restore*; and 5) *Finalization stage*. This schema is shown in Figure 2.

Coordination Stage

The coordination stage starts when the `mig` framework on the Head Node Process receives from `ras` a migration request specifying a source and a destination node.

The `mig` framework spawns an `orted-restore` daemon on the destination node, which is therefore able to receive the migrating ORTE daemon. Then, via `plm`, the framework issues a `MIGRATION_PREPARE` command to all the ORTE daemon instances running over the system, broadcasting the information related to the migration request. When the ORTE daemon instances receive the command, they notify the request to their children (the application processes) using the signals provided by Linux-OS[2].

The signal handler, implemented in the OMPI library, intercepts the `MIGRATION_PREPARE` signal/command. The `btl` TCP component of the processes that are not migrating performs the following actions: 1) caching of any future send request towards the migrating processes; 2) terminating any ongoing data transmission (`send()` system calls) towards the migrating processes; 3) flushing the transmission buffer; and 4) performing a `shutdown` system call on the transmission-side of the TCP socket.

After that, the processes send back an acknowledgement to their own ORTE daemon instances, ensuring that no further transmissions will be performed towards the frozen processes. In turn, the ORTE daemons forward the acknowledgement to the Head Node Process. This synchronization protocol, which is depicted in Figure 3, is involved in all the subsequent phases.

[2]Open MPI developers have planned to release the `pmix` framework, which allows the ORTE daemon to communicate via Unix sockets to its children. Our approach will be changed accordingly

CRIU Dump

Once all the ORTE daemon instances are aware of the migration request, the Head Node Process can issue the `MIGRATION_EXEC` command and effectively start the migration procedure. When an application process receives the `MIGRATION_EXEC` command, it waits until all the in-flight packets have been received by the destination side. At this point, all the TCP connections towards processes involved in the migration can be safely closed, and an acknowledgment can be sent back to the ORTE daemon.

When the migrating ORTE daemon receives the acknowledgment, it uses the API provided by the CRIU library to perform the checkpoint of its execution status. The checkpoint outcome, i.e. the generated process dump, is stored in a temporary directory. Following the Open MPI common practice, the temporary directory is set to `/tmp`. However, a problem may arise if `/tmp` is mounted in the main memory (most common case) and the amount of memory available is not enough to store the dump. To address this issue, the user or the resource manager can specify a different directory.

Process State Migration

As previously described, the outcome of the CRIU checkpoint (or dump), i.e. the process dump, is a collection of files. To simplify the transfer of such files over the network, the next step is to create an archive containing such files and optionally compress it. For brevity we are going to call the checkpoint archive ''image".

Generally, the decision of compressing or not the archive requires the evaluation of the trade-off between compression time and transfer time savings due to compression. This task can be in charge of the resource manager, which should consider several factors, e.g. network bandwidth, network traffic, image size, shared memory occupation (see Section 4.2) and disk performance.

The image is now ready to be moved to the destination node. This can be achieved according to two strategies: 1) using a TCP connection between source an destination node; 2) using a Network File System (NFS). At the time of writing, we do not have a storage unit with NFS available for testing, therefore we selected the TCP based option to transfer the image between nodes.

CRIU Restore

When the `orted-restore` daemon running on the destination node receives – and possibly decompresses – the image coming from the source node, it restarts the ORTE daemon and its children processes using the CRIU API. Since the C/R approach of CRIU is totally transparent to the (frozen) processes, after the restart we need to send a signal to the restored ORTE daemon to advise it that a node migration has occurred. Accordingly, the ORTE daemon reopens the connection to the Head Node Process and sends the `MIGRATION_DONE` message.

Finally, the Head Node Process broadcasts the `MIGRATION_DONE` message to all the other ORTE daemons and processes using the same synchronization protocol previously described.

Finalization Stage

When all the processes have received the `MIGRATION_DONE` message, the migration procedure enters the *Finalization*

Benchmark	Class A	Class B	Class C	Class D
IS	64	256	1024	16793
MG	128	128	1024	8294
BT	2	8	32	518
LU	2	8	32	518
SP	2	8	32	518

Table 1: Problem data sizes [MB] for each class and benchmark considered.

stage. In this phase, to minimize the overhead, the migrated processes reopen the connections towards other processes only if needed. This happens if there are packets waiting in the buffer or if the application has new data to transmit.

Once again, it is worth underlying that the entire migration procedure is performed without the awareness of the application, which only experiences a network delay in the communication towards migrating processes.

Moreover, the performance degradation is additionally mitigated by having all the nodes not involved in the migration still communicating between each other. In such a way, we avoid using complex algorithms to reach a global consistent state. This is another key advantage of our solution, since in application-level C/R schemes the coordination phase presents several problems on both user and framework sides [16].

4. EVALUATION

This section includes the results of a set of experimental tests performed to evaluate the overhead introduced by the exploitation of the proposed process migration mechanism. In this regard, we distinguished between two types of overhead: 1) performance loss due to the execution of multiple ORTE daemons on the same node; and 2) time require to actually perform processes migration.

The former overhead results from splitting the control of the application processes execution under different ORTE daemons. Accordingly, some processes cannot rely any more on shared memory to communicate with each other, even if they are running on the same node. In such cases, we need to use TPC/IP connections for inter-process communication, which has been shown to be less performing than shared memory [17]. The second overhead type instead is the time lost to complete all the migration procedure described in the previous section.

The following subsections are focused on the evaluation of each of the aforementioned overhead types. We tested the sustainability of our approach by triggering migration requests during the execution of some state-of-the-art benchmark applications on a small distributed computing system.

4.1 ORTE daemons granularity overhead

To evaluate the overhead introduced by splitting the control of the MPI processes among multiple ORTE daemon instances, we used a computing node equipped with two Intel Xeon E5-2640 octa-core hyper-threaded CPUs, with 128GB of RAM per CPU (NUMA). As common in HPC environments, we disabled Hyper-Threading, remaining with 16 cores at our disposal. The running operating system was CentOS 6.7 with updated Linux kernel version 3.18.

Benchmark	Class	# orted	Overhead %
IS	B	2	4.68
		4	5.18
		8	4.85
		16	4.85
	C	2	2.21
		4	2.68
		8	2.64
		16	2.64
	D	2	0.87
		4	0.96
		8	0.79
		16	0.64
MG	B	2	1.28
		4	4.36
		8	1.73
		16	3.24
	C	2	2.25
		4	0.62
		8	0.88
		16	1.57
	D	2	1.13
		4	1.25
		8	1.79
		16	1.50

Table 2: Static overhead of IS and MG kernel with increasing migration granularity, i.e. increasing number of ORTE daemons, compared with single ORTE daemon case.

For the experimental evaluation we executed applications from the NAS Parallel Benchmarks suite (NPB) [18]. We selected IS, MG kernels and BT, SP, LU pseudo-applications. The pseudo-applications are system solvers. The IS kernel is a balanced computation-communication benchmark, while MG is a workload focused on short and long distance communication.

The kernels have been executed specifying input classes B, C, D, which determine the problem size, as shown in Table 1. For the pseudo-applications, instead, we did not consider class D since the completion of the execution would require too much time using the systems at our disposal. Finally, we excluded the class A because the problem size would have been too small, leading to very short executions.

We executed the pairs (benchmark, class) spawning 16 processes for each benchmark execution. We selected the number of ORTE daemons controlling the MPI processes according to different granularities: 1, 2, 4, 8, 16. Each ORTE daemon instance had therefore to manage 16, 8, 4, 2 or 1 MPI processes respectively. In particular, the case of single ORTE daemon instance is a standard execution of unmodified Open MPI and we took it as a reference result in the overhead evaluation.

We measured the execution time of each tuple (benchmark, class, granularity), starting after the MPI_Init call and stopping before the MPI_Finalize call. The time needed to spawn the ORTE daemons and the processes are

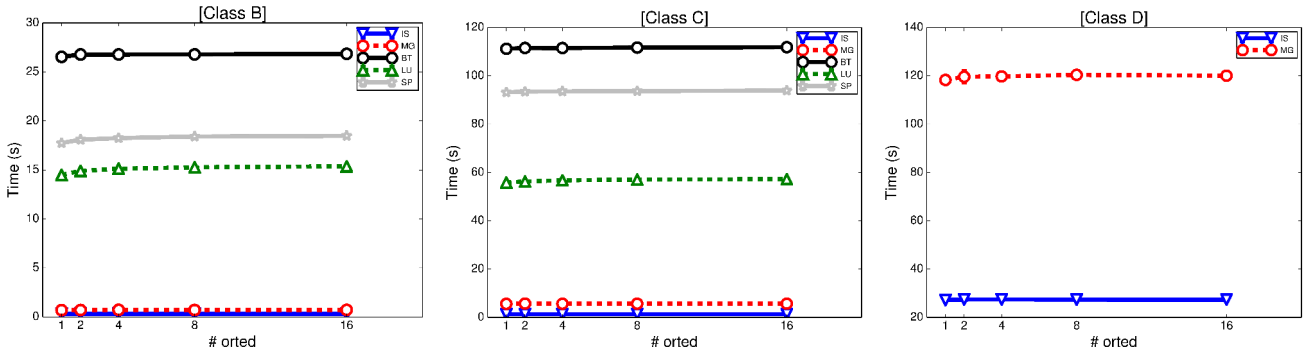

Figure 4: Execution time of each benchmark when running 16 processes using a number of ORTE daemons that ranges from 1 to 16.

Benchmark	Class	# orted	Overhead %
BT	B	2	0.92
		4	0.93
		8	0.93
		16	1.17
	C	2	0.31
		4	0.29
		8	0.45
		16	0.60
SP	B	2	1.90
		4	2.83
		8	3.72
		16	4.12
	C	2	0.31
		4	0.40
		8	0.52
		16	0.79
LU	B	2	2.92
		4	4.37
		8	5.42
		16	6.10
	C	2	0.73
		4	1.63
		8	2.19
		16	2.48

Table 3: Static overhead of BT, SP, and LU kernel compared with single ORTE daemons case.

therefore not considered. We repeated the test 20 times to obtain a significant statistics. It turned out that we experienced an average standard deviation below 1% of the total execution time.

The overall results are shown in Figure 4. The global trend is that the overhead increases sub-linearly with respect to the number of ORTE daemon instances, while it decreases as the problem size increases. The sub-linear increase of the overhead can be explained by the fact that, once there are at least two ORTE daemon instances, the TCP/IP communication between MPI processes on different instances becomes the bottleneck for communication latencies; adding more ORTE daemon instances to the existing ones does not tend to further degrade performance. The decrease of the overhead in case of increasing problem size, conversely, is due to the fact that increasing problem size means that more time is spent on computing data; therefore, the time spent in communication – which is where the overhead applies – decreases in percentage.

Looking at the data summarized in Table 2 for IS and MG, and in Table 3 for the pseudo-applications, we can state that the ORTE daemons granularity poorly affects the application execution time. Considering all the test cases, we can observe indeed that the percentage of time loss remains in the $0 - 5.x\%$ range.

4.2 Migration overhead

We characterized the migration overhead by running the benchmarks on two nodes connected via Gigabit Ethernet, both equipped with two Intel Xeon E5-2640 CPU, 128GB of RAM per CPU (NUMA) and keeping the Hyper-Threading disabled.

In the experimental migration scenario, we launched two ORTE daemon instances per node, each one managing 4 out of the 16 application processes. The resource manager triggered the migration requests after 25 seconds of execution.

To better observe the composition of the migration overhead, we divided the migration time, isolating seven contributions. We considered the time required by each of the five migration phases previously described in 3.4, plus two additional contributions: 1) the time required to encapsulate the CRIU process dump into the archive and 2) the time to extract the dump from the archive after the migration. With the exception of the *Coordination* and the *Finalization* phases, the other contributions are expected to be strongly dependent on the problem size, in particular on the image transfer time. In this regard, we evaluated the possibility of introducing the compression of the image generated by the CRIU checkpoint before proceeding with the transfer. In such a case, a decompression step is obviously required on the destination node, before resuming the execution of the processes. To this purpose, we used the GZIP compression algorithm [19].

In Figure 6, we can see how the compression is effective in reducing the size of the checkpoint image to transfer. This because of the shared memory implementation. Open MPI in facts allocates over 100MB of unused shared memory as ghost files initialized as zeros. The consequence is that compression is very effective in such cases, resulting in image

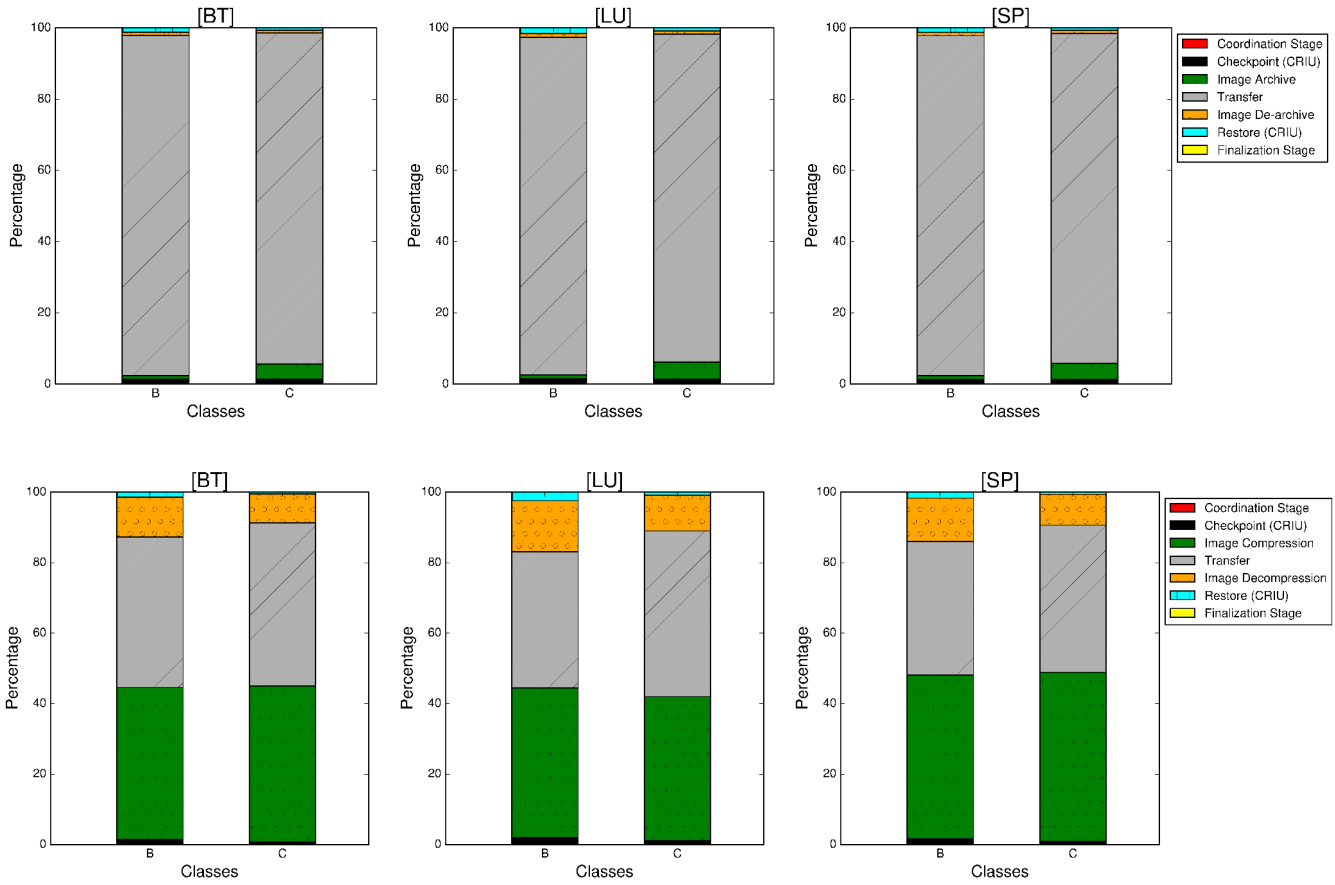

Figure 5: Process migration time composition with respect to the input problem size. Top images: migrations without image compression. Bottom figures: migration with image compression.

sizes scaled down to $22-38\%$ with respect to the original sizes. In case of bigger datasets instead – like the `C` class – this phenomenon is less evident, with compressed image sizes resulting the $45-65\%$ of the original sizes.

The composition of the migration overhead is highlighted in Figure 5. If the compression is not applied (top figures), the time required to transfer the process image over the network dominates the whole migration process ($80-90\%$ of the time) independently of the benchmark and of the class of input data. The contributions due to the synchronization stages (*Coordination* and *Finalization*) and the time needed for doing checkpoint/restore with CRIU are instead negligible.

Conversely, when the compression is applied (bottom figures), the transfer time is reduced, but a new overhead contribution is introduced. The time spent to perform image compression/decompression is not negligible. We can observe indeed an overall percentage value comparable to the transfer time ($40-50\%$) for the compression, plus a $10-15\%$ for the decompression.

Figure 7 provides an overview of the measured migration times, comparing the cases with image compression against cases where no compression is applied. For the `IS` and `MG` benchmarks, where hundreds of MB of data must be moved, the compression represents a penalty. Conversely, for `BT`, `LU` and `SP`, where data size is a few MB, break-even points can be found. Generally, we can state that resource manager should be in charge of choosing whether apply compression or not, taking into account application properties, input data size, node capabilities and network parameters (e.g., topology, bandwidth, etc...). The `mig` framework is then driven accordingly.

Please note that these tests were performed on two computational nodes connected via Gigabit Ethernet. Most recent HPC systems connect nodes using InfiniBand, which is much faster than Ethernet. It follows that, in the average case, we do not expect compression to be needed, because transfer time will usually be in the range of seconds rather than of tens of seconds. In our experimental set-up, as shown in the results, the time required to interrupt the execution of a set of processes, to migrate them to another node and to resume their execution is not negligible and may require tens of seconds.

5. CONCLUSIONS

In this paper we introduced a novel approach to support process migration in the Open MPI framework. The approach is based on handling the execution of multiple ORTE daemon instances, which can be thought of as the smallest migratable unit. This is performed transparently to the application and the non-involved Open MPI frameworks and components.

Compared to other state-of-the-art solutions, one of the major advantages of our approach is the *maintainability*.

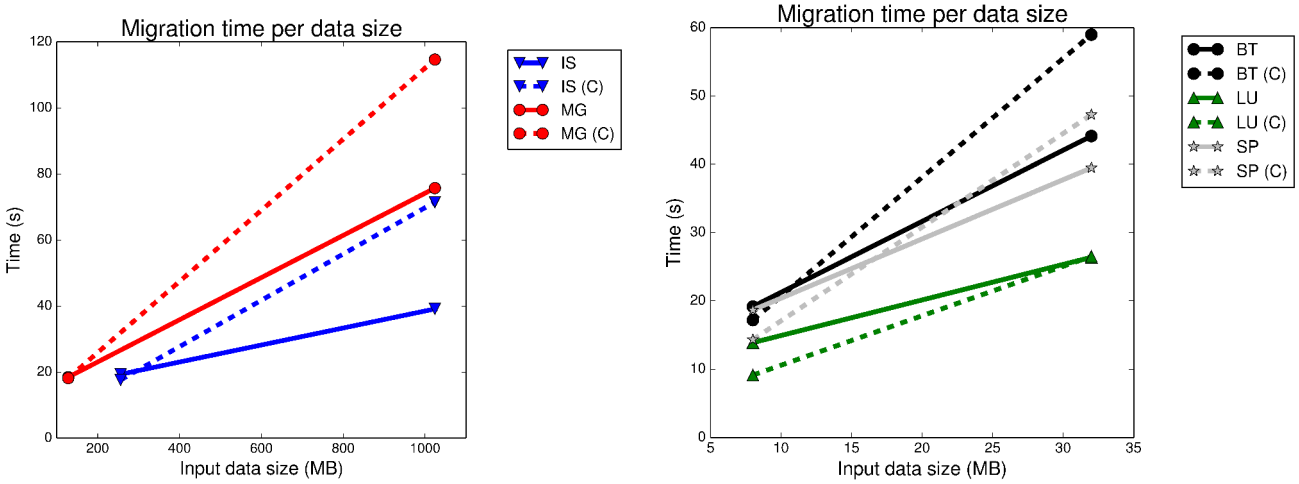

Figure 7: `IS`, `MG` and `BT`, `SP`, `LU` migration overhead with respect to the input problem size. Migrations using image compression (dashed lines (C)) can be compared to migrations without image compression.

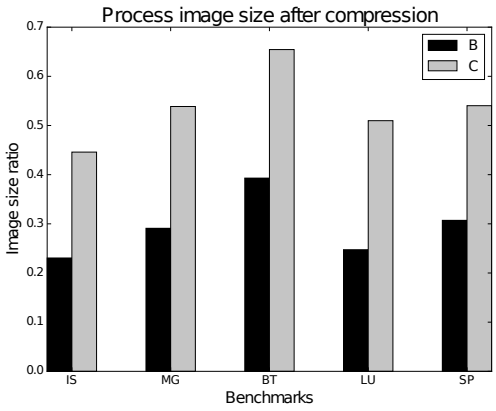

Figure 6: Process checkpoint image after GZIP compression for each benchmark, with respect to the input data class

The extension introduced in the Open MPI runtime in fact has a minimal impact on the other Open MPI frameworks. Furthermore it does not require any change on the applications code.

Moreover, the `mig` framework does not rely on any virtualization layer. This constitutes a gain in terms of performance with respect to approaches based on virtual machine allocation. In this regard, our proposal allows us to perform fine-grained migrations, since the resource manager can dedice between migrate an entire application or a subset of its processes. This feature also increases the controllability of the workload execution.

Additionally, a third party agent (a resource manager) can be the entity responsible of controlling the migration at system-level, hence lifting the application from the burden of inferring conditions for which a migration is worth considering.

Through experimental tests, we shown how the overhead due to grouping the application processes on top of several ORTE daemons can be considered negligible. Conversely, stopping and resuming the processes execution on different nodes, introduces an overhead dependent on the specific application, its input data size, the network and the node capabilities. As a consequence, a resource manager can play a key role to evaluate when a migration is worth to be performed.

The major limits of the proposed process migration mechanism are in similar to those of the other C/R based systems: the nodes of HPC systems must be homogeneous, i.e. the operating system (with kernel version), libraries version and application binaries must be perfectly identical. Moreover, performing the checkpoint with *CRIU* requires administration level permissions (`root` user in Linux) in all the nodes. As a future work, we may allow ORTE daemons to spawn application processes without such a requirement.

From the MPI communication standpoint, the lack of *InfiniBand* support is currently the most important missing feature. However the development of this component is currently ongoing.

Overall, we can state that the work presented in this paper is the first process-level migration feature developed for Open MPI whose control is kept at system-level (resource manager) and that does not require the code of applications to be changed.

We propose this migration framework to enable new flexible resource management and fault-tolerance strategies on HPC systems running Open MPI based applications. In this regard, we aim at developing future works using the *Barbeque Run-Time Resource Manager (BarbequeRTRM)* [20] to implement policies exploiting the `mig` framework.

6. ACKNOWLEDGMENTS

This work was supported in part by the EC under the grant HARPA FP7-612069[3] and within the framework of European Union funded project with reg. numbers CZ.1.07/2.3.00/30.0055, CZ.1.05/1.1.00/02.0070.

We also thank Open MPI and CRIU communities for the very valuable assistance and collaboration, and IT4Innovations

[3]http://www.harpa-project.eu/

(IT4I), Ostrava (CZ)[4] for providing us the systems for performing the experimental evaluations.

7. REFERENCES

[1] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[2] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 43. IEEE Press, 2008.

[3] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[4] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[5] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[6] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[7] Joshua Hursey, Timothy I Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in open mpi. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 49–58. ACM, 2009.

[8] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[9] Miguel G Xavier, Marcelo Veiga Neves, Fabio D Rossi, Tiago C Ferreto, Tobias Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[10] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[11] Simon Pickartz, Jens Breitbart, and Stefan Lankes. Impacts of virtualization on intra-host communication. 2016.

[12] Criu - checkpoint/restore in userspace. https://criu.org/. Accessed: 2016-04-11.

[13] W. Li, A. Kanso, and A. Gherbi. Leveraging linux containers to achieve high availability for cloud services. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 76–83, March 2015.

[14] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[15] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 2013.

[16] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

[17] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[18] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[19] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[20] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):39, 2015.

[4] https://www.it4i.cz/