Supporting Automatic Recovery in Offloaded Distributed Programming Models Through MPI-3 Techniques

Antonio J. Peña
Vicenç Beltran
(antonio.pena,vbeltran)@bsc.es
Barcelona Supercomputing Center (BSC)

Carsten Clauss
Thomas Moschny
(clauss,moschny)@par-tec.com
ParTec Cluster Competence Center GmbH

ABSTRACT

In this paper we describe the design of fault tolerance capabilities for general-purpose offload semantics, based on the OmpSs programming model. Using ParaStation MPI, a production MPI-3.1 implementation, we explore the features that, being standard compliant, an MPI stack must support to provide the necessary fault tolerance guarantees, based on MPI's dynamic process management. Our results, including synthetic benchmarks and applications, reveal low runtime overhead and efficient recovery, demonstrating that the existing MPI standard provided us with sufficient mechanisms to implement an effective and efficient fault-tolerant solution.

ACM Reference format:
Antonio J. Peña, Vicenç Beltran, Carsten Clauss, and Thomas Moschny. 2017. Supporting Automatic Recovery in Offloaded Distributed Programming Models Through MPI-3 Techniques. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079093

1 INTRODUCTION

OmpSs is a popular programming model (PM) for high-performance computing (HPC) based on compiler directives and task decomposition. It incorporates functionality to ease the programmer's management of resources efficiently, hence greatly fostering programming productivity and maintenance. Recently, *collective offload* extensions have been proposed for their incorporation in this PM. Apart from mapping greatly to some algorithms, these ease the efficient use of heterogeneous compute nodes by enabling the offload of tasks to compute nodes featuring the most suitable architecture.

The nature of the distributed offload semantics makes them especially vulnerable to a variety of local failures that may well propagate to the entire application causing its abnormal termination. Our proposal is focused in protecting from fail-stop failures ultimately causing a process to lose communication with its peers, such as process abortions or hardware failures. Traditional approaches based on checkpoint/restart (C/R) may be used to mitigate this situation and prevent having to restart the execution entirely. The specific semantics of this PM, however, lead us to investigate on more efficient fault tolerance capabilities. By basing our (partial) restarting techniques in PM semantics instead of user-specified or periodic checkpoints, we provide lower runtime overhead and efficient recovery. Although we use OmpSs offload to showcase our idea, our approach should be applicable to other similar offload PMs such as [8].

The Message Passing Interface (MPI) is a *de facto* standard in HPC communication. Offering a common application programming interface (API), the different hardware vendors provide highly-optimized MPI implementations for their platforms, making MPI a good choice to leverage efficient portable communication. The OmpSs collective offload feature has been designed to interoperate with MPI applications; internally, it relies on the MPI process spawning capabilities.

In spite of the efforts of the community, the MPI Forum has not yet incorporated specific fault-tolerance capabilities into the MPI Standard. The few references to error detection and reporting in the latest release (version 3.1) are often ambiguous, but also optional to adopt by the MPI implementors. Hence, users requiring fault-tolerance capabilities in MPI applications are often forced to either implement application-dependent resilience mechanisms or to use non-standard MPI features offered by some implementations.

Instead, in order to obtain the base MPI resilience capabilities that our solution demands, and following the original MPI philosophy which leaves fault tolerance as "a property of an MPI program coupled with an MPI implementation" [23], we have opted for incorporating error detection, reporting, and handling features into the ParaStation MPI implementation that fully comply with the MPI standard. At the OmpSs runtime level, we leverage these features to implement the detection of a failure in a remotely offloaded task and re-execute the appropriate dependent tasks on a new set of sane compute nodes in cooperation with the global resource manager. This is handled completely transparently to the user, thereby preventing application developers from having to implement complex failure handling code.

In this work, (1) we introduce MPI-3 standard compliant techniques to be adopted by MPI implementations to enable runtimes leverage resilient distributed offload semantics; (2) built on top of these, we present the design and implementation details of efficient and transparent resilience support for the OmpSs offload semantics in the Nanos++ runtime; and (3) we provide an in-depth performance evaluation of the runtime overhead and recovery efficiency of our proposal. To

ICS '17, Chicago, IL, USA
 978-1-4503-5020-4/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079093

the best of our knowledge, this is the first time the suggestion of [23, Section 5.2] to use MPI dynamic process management for fault-tolerance purposes is explored (1) showing an application domain on which this approach is practical and manageable and (2) devising the required support from the MPI implementation. Our evaluations, involving both benchmarks and applications, reveal that this proposal introduces low runtime overhead and provides efficient error recovery. In times in which the existing MPI fault-tolerance support is highly debated, we provide a use case in which the current specifications already incorporate sufficient mechanisms to implement an efficient and effective fault tolerant solution.

2 BACKGROUND

OmpSs is a directive-based PM that enables parallelism in a data-flow way. The developer is in charge of decomposing the code into *tasks* and identifying their data dependencies. This information is used by the source–to–source Mercurium compiler to generate the corresponding calls to the Nanos++ runtime API. Nanos++ is responsible for scheduling and executing the annotated tasks, preserving the implied task dependency constraints. Further general information can be found at http://pm.bsc.es.

The dynamic offload functionality recently incorporated into OmpSs [10] enables the execution of tasks in remote compute nodes. Apart from providing enhanced support for node heterogeneity (tasks may be offloaded to those compute nodes featuring the most suitable architecture), the offload semantics map greatly to some algorithms, yielding programmability benefits. Unlike other offload-based PMs, OmpSs is designed to offload full general-purpose tasks. This model introduces in OmpSs the concept of *master* and *booster* nodes/processes, where the former execute the main dataflow and the latter the offloaded tasks. A *booster* process may only execute a task instance at a time, but users may allocate an arbitrary number of *boosters* per host. The OmpSs offload extensions allow recursivity, enabling *booster* nodes act as *masters* of their offloaded tasks. Data transfers are handled implicitly from the task data dependency information.

The offload extensions consist of *booster* allocator and releaser functions plus a new clause for task offloading. The allocator (`deep_booster_alloc`) internally calls `MPI_Comm_spawn_multiple`, returning the created intercommunicator that *master* processes may use to interact with their *booster* processes. It also gets an intracommunicator to specify the set of spawning processes. Figure 1 depicts an example of 4 MPI *master* processes allocating 2 *boosters* on the `MPI_COMM_WORLD` communicator which in turn allocate 3 *booster* processes each (i.e., using the `MPI_COMM_SELF` communicator). The new `onto` clause is used in combination with the `task` clause to offload a particular task to a given *booster* process identified by its intercommunicator and MPI rank number.

3 RELATED WORK

This section reviews prior work on the topics related with the specific contributions of this paper. A discussion about how

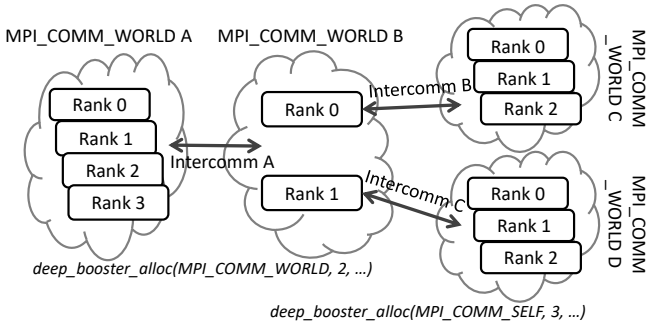


Figure 1: OmpSs offload sample process structure.

the OmpSs offload mode is related with other PMs featuring offload semantics (such as Intel Offload [8], CUDA [25], OpenCL [24], OpenACC [26], or OpenMP 4.0 [27]) or targeting cluster-level heterogeneity (VCL [1], rCUDA [4, 28], MPI spawn) can be found in [10].

3.1 Checkpointing

The traditional mechanism to protect application data has been C/R: data is saved regularly so that it can be restored upon failure. For instance, FTI [16] and SCR [5] are popular checkpointing libraries. In contrast, our proposal ensures that remotely offloaded tasks finalize properly in spite of eventual connection lost events, preventing from the need to restart the entire application from a past state. Both solutions are fully compatible and complementary to each other and their combination should decrease checkpoint frequency.

3.2 MPI Resilience

Up to the current version of the MPI Standard (3.1), there is no specific support for fault tolerance capabilities. Although it contemplates the possibility that MPI functions do not cause a program abortion upon a failure by setting arbitrary error handlers, continuing a program in which an MPI call failed is not guaranteed to be possible. MPI implementations, however, are not forbidden to support program continuation after properly handled errors at their will. For instance, the Hydra process manager of MPICH features a command-line option (`--disable-auto-cleanup`) to prevent killing all the processes upon any abortion [3]; however, this leaves live processes with broken communicators. We develop novel standard-compliant fault tolerance capabilities into the ParaStation MPI implementation [30] to control process cleanup and provide the required error reporting.

The MPI community has been working on providing specific support for enabling communication among live processes after one or several processes of a communicator failed. User-Level Failure Mitigation (ULFM) [22] is the proposal currently being discussed and iteratively refined at the MPI Forum. Although we can find implementations supporting it experimentally, it cannot be considered MPI compliant, and its contents are likely to change before its possible final incorporation into a future version of the Standard. Several

other works have proposed extensions to the MPI API for incorporating a variety of fault tolerance capabilities, but no one has been accepted into the MPI standard so far. Automatic and semiautomatic fault tolerance within the MPI implementation (i.e, without user intervention) have been studied, but proven to pose high overhead at scale [11, 17].

3.3 Resilience in Task-Based PMs

Prior to the incorporation of the OmpSs offload functionality, "smart" C/R was introduced in the Nanos++ runtime to provide efficient fault-tolerance capabilities by benefiting from the PM semantics (leveraging the task data dependencies) [19]. This protected from memory faults reported by the OS. Our proposal is complementary to this solution, targeting specifically the recently-introduced offload semantics.

The NABBIT task graph scheduling framework was added fault tolerance capabilities in [18]. This and other previous works leveraged local task reexecution as a response to transient errors. Recovering from soft errors supporting distributed task graphs was considered within the PaRSEC task-based runtime framework [7], not including offloaded semantics—posing high storage overheads and possibly a long list of predecessors to be reexecuted to recover damaged data—nor *connection lost* events. Our work, however, focuses specifically on adding automatic fault tolerance capabilities to *distributed offloading semantics*, covering from the MPI-level error detection and recovery to the upper-level runtime implementation and associated PM semantics.

The CIEL execution engine for cloud environments provided fault-tolerance to a conceptually similar task-based PM involving masters and workers [9]. CIEL does not use MPI underneath nor considers collaborative offloaded tasks and hence limits its approach to heartbeat monitoring and sane worker reexecution. In fact, that paper mentions application C/R as the only possibility to attain resiliency in similar HPC environments leveraging MPI. Our work demonstrates that a more efficient approach is possible.

3.4 Resilience for Offload-Based PMs

The most popular offload-based PMs are those targeting accelerators. Resilience has been provided leveraging redundant computations [14]. Hauberk [15] is able to restart a GPU application from a checkpoint upon a failure detected by automatically-inserted silent data corruption detectors.

CheCUDA [12] and CheCL [13] are C/R solutions for CUDA and OpenCL applications, respectively. Snapify provides C/R, migration, and swapping services for Intel® Xeon Phi™ coprocessors [6]. These do not target specifically coprocessor failures nor integrate fault detection mechanisms.

VOCL-FT [29] benefits from OpenCL's offload semantics to provide a transparent and more efficient microcheckpointing and partial restart mechanism inside the OpenCL implementation. Although the proposed optimization techniques are not OpenCL specific, these are meant to be efficient for offload-based PMs with fine-grained, iterative kernel offloads (with their corresponding data movements between host and accelerator memories). Since the OmpSs offloading mode targets remote general-purpose processing platforms, it is expected to leverage *coarse-grained tasks*, hence minimizing the data transfer overhead across a network. Therefore, our work does not propose using a checkpoint-based recovery of the offloaded processor's memory space, hence reducing runtime overhead.

4 SUPPORT FROM THE MPI STACK

We have developed additional capabilities compliant with the MPI 3.1 standard into ParaStation MPI for providing the features needed by the upper-level distributed runtimes. ParaStation MPI is an open-source MPI library developed and supported by ParTec GmbH together with the Jülich Supercomputing Centre (JSC). ParaStation MPI, which is in turn based on MPICH, is fully MPI 3.1 compatible and offers support for a multitude of HPC-related networks.

4.1 Process Manager

The process management framework of ParaStation MPI (called *psmgmt*) is implemented in the form of an efficient and robust network of control daemons. One instance of such a daemon (called *psid*) is running on each node of the system and all the distributed daemons are then linked together by means of a highly-scalable communication subsystem. This subsystem is based on a Reliable Datagram Protocol (RDP) that is used for inter-daemon signaling as well as for I/O and signal forwarding with respect to the actual MPI processes [2].

Each node-local daemon instance is also responsible for creating the local processes belonging to a distributed MPI session. After forking, the *psids* keep control over the MPI processes and constantly monitor their existence as well as the responsiveness of the other *psids* in the network. If an MPI process should fail, then the local *psid* detects this event and notifies all the other daemons for cleaning the whole MPI session belonging to the failed process. In addition, if a whole node along with its local *psid* should fail, the daemon network detects the absence of corresponding heartbeat messages of the deceased *psid* and can react accordingly. That way, a proper cleaning and release of all resources still allocated by the failed session is guaranteed.

For the handling of larger systems, *psmgmt* may be combined with an outer and more generic resource manager featuring a batch queuing system together with a job scheduler like TORQUE/MAUI or SLURM. In doing so, *psmgmt* follows a strict *single-daemon* concept where the native monitoring daemons of the resource managers are replaced by special plugins for the *psids*. That way, the *psids* can directly interact with the outer resource manager and report a failed MPI session, e.g., for aborting the whole related job.

However, for providing fault tolerance, less radical but more sophisticated measures have to be taken in order to keep healthy process groups alive for eventually recovering the faulty session parts. Specifically, the process manager had to be extended with the ability to differentiate between the processes sharing the same `MPI_COMM_WORLD` communicator

and those not belonging to the same process group but to the same session. That way, the *psids* are now able to clean, e.g., a group of failed *booster* processes while keeping the *master* processes alive, which may then strive for a respawn in a sane set of nodes for recovering the failed offloaded tasks.

The logic for this process group awareness is naturally rooted in the `MPI_Comm_spawn(_multiple)` call. When a program enters this function, a dialog between the respective MPI process and the local *psid* is conducted via the Process Management Interface (PMI) [20]. During this dialog, the needed information for creating and interconnecting to the newly spawned process group is exchanged.

By utilizing the `MPI_Info` object, as it is to be passed as an argument of the spawn function, the *master* processes can tell the *psids* if they explicitly want to be treated independently from the child processes forming the *booster* group. For doing so, the root process of the *master* group calling the spawn function has to add a (`key="parricide"`, `value="disable"`) pair to the *info* object. As a result, a process fault in the child group triggers a cleaning of the *booster* processes only, whereas the *masters* are kept alive and running. However, this behavior is consciously implemented asymmetrically: a process fault in the parent group still results in a cleaning of *all* processes belonging to the affected session.

4.2 Communication Layer

The lower-level communication layer of ParaStation MPI (called *pscom*) is specially designed for its employment in HPC systems. As such, a variety of interconnects and interfaces commonly used in this specific application field is supported by *pscom*. For doing so, the library exhibits a flexible architecture featuring *plugins* for all the different interfaces and protocols. Such plugins are loaded and selected at library runtime by means of a priority/fallback scheme, which means that plugins promising faster communication than others are preferred while slower but more robust counterparts may still be used in case of an unsuccessful initialization of the former.

As the lowest common denominator, socket-based communication via the TCP/IP protocol serves as a sort of *pseudo*-plugin, which provides a scalable on-demand connection establishment by delaying the actual connection setup to the first send request posted for the respective peer. That means that all needed connections are initially made via TCP/IP and that other plugins may then use these socket channels for exchanging further information like keys, queue pair numbers, and other identifiers as needed, for instance, for establishing InfiniBand-based communication.

Usually, these initial socket connections are directly closed after the initialization of a higher prioritized plugin. However, for providing fault tolerance, these may also be kept intentionally open in order to enable an OS-assisted detection of broken links. In doing so, the still open socket connections are handed over to dedicated *connection guard* threads waiting in `select` calls for detecting TCP events, whereas the regular MPI communication among the processes is then normally conducted via the connections of a higher prioritized plugin like one of those for InfiniBand or Extoll.

Such TCP events could either be the arrival of an *end of file* message, indicating the proper shutdown of a connection, or the occurrence of an error, for example, triggered by the absence of a *TCP keep alive* message (i.e., a TCP timeout). By queuing and forwarding such events from the connection guard threads to the main threads of the MPI processes, the latter can eventually analyze and transform the events into appropriate error codes for the application layer. Since the connection guard threads commonly block most of the time within the `select` function, their impact onto the regular MPI communication performance should practically be negligible.

4.3 Discussion on Standardization

As mentioned in Section 1, we follow the original MPI fault-tolerance philosophy. In case we would like such a solution to operate on *any* MPI implementation, the MPI Standard should (1) incorporate a new reserved MPI_Info key as described in Section 4.1; (2) require implementations honor reserved keys; and (3) enforce reporting broken connections as returned error codes in MPI calls (see Section 4.2).

On the other hand, the MPI Forum is currently discussing several minor modifications to the Standard affecting fault behavior, like limiting the effect of failures to process groups leveraging involved communicators or distinguishing catastrophic error codes (see issues #1, #3, and #28 on http://github.com/mpi-forum/mpi-issues/issues). Our solution would benefit from these proposals being standardized, preventing the use of a custom *info* key.

5 OMPSS RESILIENT OFFLOAD

In this section we first discuss the high-level design of our approach in terms of functionality and semantic constraints. Next, we provide insight on implementation details.

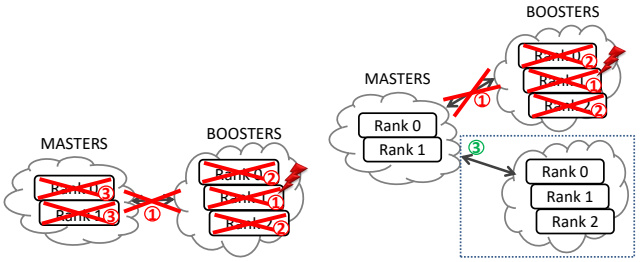
5.1 Functionality

Our main goal is to provide automatic resilient OmpSs offload functionality, so that the failure handling burden is removed from the application level. For example, the snippet of code shown in Figure 2 would allocate `ppn` *booster* processes in each of `n` hosts, offload a task to each *booster*, and wait for their execution finalization, relying on the Nanos++ runtime for the proper execution of the offloaded tasks in case of an eventual failure ultimately leading to the interruption of interprocess communication. If root *master* node failures were a concern, these could well be addressed additionally leveraging C/R. Since the resiliency feature involves semantic side effects as explained in Section 5.2 and it may pose some performance impact, we require users to specify the new `recover` clause to leverage this functionality.

The OmpSs offload primitives are designed to interoperate with MPI applications and internally use the MPI infrastructure to *spawn booster* processes. By default, MPI process managers terminate all processes upon the abortion of any of them, as depicted in Figure 3a. Since this action is performed

```
// Allocates 'ppn' processes in each of 'n' hosts
deep_booster_alloc(MPI_COMM_WORLD, n, ppn, &worker);
for (int i=0; i<n*ppn; i++) {
    #pragma omp task onto(worker, i) recover
    offloaded_task();
}
#pragma omp taskwait
deep_booster_free(worker);
```

Figure 2: Sample offloaded application pseudocode.



(a) Default process cleanup. (b) Smart cleanup+recovery.

Figure 3: Process cleanup upon failure with OmpSs offload. Failure originated at *booster* rank 1. Numbers indicate possible failure propagation order.

at process manager level, MPI error handlers are not involved and hence cannot prevent process termination. Our desired behavior, however, is that the MPI process manager would limit the process cleanup upon failure to those processes sharing the same `MPI_COMM_WORLD` communicator of the failed process. As shown in Figure 3b, our idea is to replace the failed *booster* processes with a freshly spawned set of them.

With the support from the MPI environment discussed in Section 4, the Nanos++ runtime will be able to detect a failed group of *boosters*, provide a replacement for these, allocating a new set of *booster* processes in sane nodes (we implement the integration with the resource manager to incorporate new resources described in [21]), and restart the required operations. Nothing prevents recoverable offloaded tasks from containing further subtasks; a failure in a *booster* process would trigger the termination and replacement of all sibling and descending *booster* processes.

Heterogeneous hardware, either in form coprocessors (i.e., GPUs) or heterogeneous compute nodes, is fully supported. Data copies present in the failed booster node at the beginning of the task (including those in accelerator memories) are invalidated in the *master*'s Nanos++ runtime upon failure. Copies of the input data are already stored by design by the Nanos++ runtime of the *master* processes and sent as part of restarting the task.

5.2 Semantic Constraints

The original semantics of the OmpSs offload capabilities enable *master* processes to communicate directly with their *boosters* performing MPI calls by employing the intercommunicator provided by the `deep_booster_alloc` call. An internal

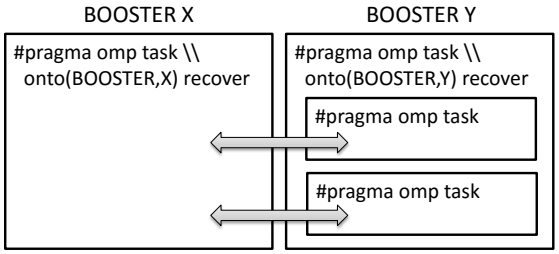


Figure 4: Subtasking to define a `recover` domain.

replacement of the original intercommunicator upon failure and recovery, however, would initially prevent this feature. The MPI profiling interface could be used to intercept MPI calls and perform the proper communicator mapping. Since we have not found any use case for this feature, however, we chose to disallow this practice for tasks with recovery capabilities, considering the value returned by the *booster* allocation call an opaque handler to identify the set of *boosters*. In the end, data transfers between *masters* and *boosters* should be expressed leveraging the OmpSs semantics by means of *data dependencies* and performed internally by the runtime.

Since tasks offloaded to the same set of *boosters* may communicate using MPI operations assuming the corresponding peers to be present, we need to identify which tasks to restart upon the failure of one of them. For instance, some tasks presenting communication dependencies among themselves may have already finished properly when a failure is noticed in one of their peer tasks. On the other hand, a big task may be designed to communicate with several smaller sequential tasks executed on a different *booster*. In order to guarantee that the appropriate tasks are restarted, we (1) prevent *boosters* from executing new tasks until all other *boosters* finished properly (in a "waiting for clearance" status[1]); and (2) upon failure, we restart all currently executing tasks and those waiting for clearance. If a task presenting a *communication dependence* is not yet in execution when a failure is found in another task featuring the same communication dependence, it will start on the new set of sane *boosters*, meeting the communication dependence. If several sequential tasks are designed to meet a communication dependence with a single task running on a different *booster*, these must be grouped as subtasks under a single `recover` task, as depicted in Figure 4. Hence, we leverage implicit *recover domains* by preserving the occurrence order of *recover* tasks and their natural synchronization mechanisms.

Defining explicit recover domains is also allowed to enable meeting the previous constraints where implicit synchronization mechanisms would not guarantee the proper automatic differentiation. Figure 5 shows an example of offloaded tasks, where those within the different `for` loops belong to different recover domains. The first two offloaded tasks will not release their corresponding executing *booster* until ensuring that their peer finished successfully. Specifying explicitly a

[1]Output transfers to `inout` buffers are delayed, avoiding the need of maintaining separate backup copies (checkpointing).

```
// Allocates 3 processes in different hosts
deep_booster_alloc(MPI_COMM_WORLD, 3, 1, &worker);
for (int i=0; i<2; i++) {
    #pragma omp task onto(worker, i) recover(a)
    offloaded_task();
}
for (int i=0; i<3; i++) {
    #pragma omp task onto(worker, i) recover(b)
    offloaded_task();
}
deep_booster_free(worker);
```

Figure 5: Code defining two *recover domains*.

recover domain is required if the task executing on *booster* 2 has communication dependencies with other tasks launched as part of the same loop. *Otherwise, if explicit recover domains are not specified* and the task starts execution before those belonging to the first domain finished, these would wait for their third peer to finish properly while the latter may be waiting for communication with its peers from domain `b`, causing a deadlock. Nevertheless, this situation is unlikely and usually the natural synchronization mechanisms will separate the different recover domains, so users are rarely required to manually specify recover domains. For example, the code shown in Figure 5 would likely require a `taskwait` separating both loops to prevent communication interference, or issuing both set of tasks on a different allocation of *boosters*, removing the need to manually specify the recover domains.

Last, the full or partial reexecution of a *recover* task should not pose undesired side effects.

5.3 Implementation Details

The OmpSs Nanos++ runtime leverages an *idle loop* during which the finalization of offloaded tasks is queried. Depending on the configuration of the runtime, this may be performed by a dedicated thread. The original task finalization query implementation consisted in calls to `MPI_Iprobe` to detect the arrival of finalization messages (identified by a specific MPI tag) from *boosters*. We replaced these MPI calls by calls to `MPI_Testany`. The required array of MPI request handlers is previously returned by `MPI_Irecv` calls matching the appropriate finalization tag and origin in every *booster*, initially populated during the `deep_booster_alloc` call, and reissued after every request completion. This new approach lets us determine process finalization more efficiently by avoiding the use of the unexpected message queue of the MPI implementation, but also to assess whether a *booster* process failed: in case a *booster* process is no longer connected, the improved MPI implementation will issue the corresponding request completion and error return.

This mechanism allows every *master* process notice the abnormal termination of any *booster* regardless of whether or not it had offloaded any task to the failed *booster*. However, when *booster* processes are allocated in collective communicators (i.e., not `MPI_COMM_SELF`), a *master* may notice the correct finalization of an offloaded task before another task belonging to the same domain fails. Hence, we implement a *failure consensus* mechanism consisting of `MPI_Allreduce` calls, placing properly finalized tasks on a "waiting for clearance" status until the MPI collective communication is finished and the execution is either globally cleared or determined to have failed. This failure consensus mechanism is also implemented as part of the `deep_booster_free` call to make *master* processes not currently involved in task offload execution aware of possible failures and have them participate in the collective MPI spawning call to generate a sane set of *booster* processes. Although this procedure poses tighter synchronization constraints than the original nonresilient implementation, since offloaded tasks should be coarse grained, we do not expect high overheads. The failure consensus mechanism is *not activated* for tasks offloaded to *boosters* allocated within the `MPI_COMM_SELF` communicator. Output data is flushed back to the *master* at the end of every task to ensure the most recent copy is available for a potential restore.

In the event of an error detection, the following response steps are performed to recover from the failure and resume the normal execution of the application: (1) clean data structures pointing to failed resources; (2) perform an MPI spawn and set `MPI_ERRORS_RETURN` as the error handler on the new intercommunicator; (3) map old to new communicators to properly interpret subsequent `onto` clauses / `deep_booster_free` calls; (4) reset the failed *work descriptor* (a Nanos++ internal structure representing a task instance) and associated resources to their initial state; and (5) launch the failed work descriptor on the new *booster*.

Note that we set the `MPI_ERRORS_RETURN` error handler on intercommunicators with the *booster* processes in order to prevent *masters* from being terminated by the MPI process manager upon a failure involving that communicator.

The overall failure detection and recovery implementation is summarized in pseudocode in Figure 6.

6 EVALUATION

In this section we provide an in-depth evaluation of our solution. We analyze our results based on benchmarks and applications. Unless otherwise stated, we present the average wallclock time of five repetitions of the experiment. Following the approach in [29], to provide a worst-case scenario, failures are injected hard-coding `abort` system calls right before the end of the indicated task. Although this implies a FIN packet is sent immediately by the operating system (OS), favoring failure detection time, relatively-frequent TCP keep alive messages are advised to reduce response time in case of an OS/hardware failure, yielding similar runtime overhead and recovery performance. Also, following the approach described in [10], all compute resources to be used by dynamically-created processes are available in a set of system-wide spare nodes. In the plots, `Original` refers to pragmas not employing the `recover` clause, disabling the resilience feature. The rest of the configurations do use the `recover` clause. `Error-Free` refers to executions not impacted by a fault. `Error` configurations state that an error was injected.

We perform our experiments in a 128-node homogeneous cluster (note that leveraging a heterogeneous platform would

```
recover() {
    deep_booster_free(old_comm, respawn=true)
    new_comm = respawn()
    map(old_comm, new_comm)
    for (i=0; i<size; i++)
        wd[i].reset()
        offload(wd[i])
}

testFinished() {
    res = MPI_Testany(taskEndMpiReqs, &completed)
    if (completed && currWD.isRecoverable())
        finished++
        failure = failure || res != MPI_SUCCESS
        if (finished == running)
            finished = 0
            if (shared)
                MPI_Allreduce(&failure, MPI_MAX)
            if (failure == 1)
                failure = 0
                recover()
            else for (i=0; i<size; i++)
                finish(wd[i])
        // else task is waiting for clearance
}

deep_booster_free() {
    ...
    if (!respawn)
        failure = -1
        MPI_Allreduce(&failure, MPI_MAX)
        // -1 means everybody is here
        while (failure != -1)
            if (failure == 1) respawn()
            failure = -1
            MPI_Allreduce(&failure, MPI_MAX)
    ...
}
```

Figure 6: Nanos++ failure detection/recovery.

Table 1: Connection Guard benchmarking. Max. STD: 0.04s.

Configuration		Benchmark	
Guard	Error Handler (MPI_ERRORS_)	PingPong	PingPing
Disabled	ARE_FATAL	1.455 µs	1.425 µs
Enabled	ARE_FATAL	1.444 µs	1.425 µs
Enabled	RETURN	1.452 µs	1.426 µs

only impact the execution efficiency of the offloaded tasks). Each node is equipped with two 8-core Intel® Xeon® ES-2680 CPUs running at 2.7 GHz and 32 GB of RAM. The interconnection network is an InfiniBand QDR. We implemented our functionality on top of ParaStation MPI 5 and OmpSs v15. All the code is compiled using the GNU C 5.3 compiler.

6.1 MPI Connection Guard

To assess the impact of the *pscom* connection guard feature, we used the PingPong and PingPing benchmarks from the Intel® MPI Benchmarks suite version 4.1 with their default setting for latency measurements: 1000 repetitions and zero-byte messages. Each of both benchmarks was run 10 times for each of the following scenarios: connection guard disabled, connection guard enabled, and connection guard enabled plus `MPI_ERRORS_RETURN` as the error handler. The results, shown in Table 1, confirm that the connection guard feature does not pose any noticeable performance impact.

```
deep_booster_alloc(MPI_COMM_WORLD, sz, 1, &boosters);
#pragma omp task onto(boosters, rank) recover
    puts("Hello, world!");
#pragma omp taskwait
```

Figure 7: Nanos++ offload resilience benchmark code.

Table 2: Nanos++ offload resilience benchmarking results.

# of Nodes	Original	Error-Free	Error
2	0.001 s	0.035 s	0.036 s
4	0.001 s	0.013 s	1.397 s
8	0.001 s	0.055 s	1.431 s
16	0.001 s	0.061 s	2.370 s

6.2 OmpSs Offload Microbenchmarking

We performed microbenchmarking (see Figure 7) in order to provide an initial assessment of the performance impact of our fault-tolerant OmpSs offload implementation. In a set of 1–8 *masters* leveraging the same number of *boosters*, we measured the execution time of an offloaded task using the `Original`, `Error-Free`, and `Error` configurations. The `recover` clause in Figure 7 is not included in `Original`. Our results, featuring a maximum relative standard deviation (RSD) of 27%, are shown in Table 2. These reveal up to 61 ms overhead for the `Error-Free` case—which should be negligible for coarse-grained tasks—mainly led by the failure consensus mechanism. The `Error` case poses up to 2.4 s overhead, mainly caused by the MPI spawning call (see [10]).

6.3 N-Body Benchmark

N-Body simulates the movement of a group of "bodies" (may be from particles to planets) given the forces driving their physical interaction. The dataset is usually distributed among compute nodes due to its size. At each time step, each process first calculates the interactions among its own bodies. Next, the bodies are exchanged in a ring fashion using a temporary buffer. Then, the interactions including the new bodies are computed again. This process is repeated until all bodies have visited all processes, what concludes a time step.

We note that the code developed for this benchmark is not intended to be an efficient N-Body implementation. Instead, the purpose of this benchmark is to showcase the behavior of the resilient offload capabilities both in favorable and unfavorable cases (a test case using a production application is analyzed in Section 6.4). Figure 8 shows an OmpSs implementation in which all major tasks are offloaded to a *booster* within a single *global* task. We refer to this implementation as *global*. On the other hand, Figure 9 describes an implementation in which the different tasks are offloaded separately. We refer to this implementation as *partial*. While the former constitutes a good case for the resilient offload feature because of minimizing synchronization among *boosters*, the latter does the opposite, posing a bad-case scenario: *booster* processes have to exchange more often failure information, delaying the finalization of more tasks, and multiplying the

```
void solve_nbody(...) {
    ...
    deep_booster_alloc(MPI_COMM_WORLD, n, 1, &boosters);
    #pragma omp task onto(boosters, my_rank) ...
    for (timestep=0; timestep<ntimesteps; timestep++) {
        remote = local;
        for (rank=0; rank<n; rank++) {
            #pragma omp task ...
            calculate_forces(...);
            #pragma omp task ...
            exchange_particles(...);
            remote = tmp;
        }
        #pragma omp task ...
        update_particles(...);
    }
    #pragma omp taskwait
    deep_booster_free(boosters);
}
```

Figure 8: Offloaded N-Body—*global* version.

```
void solve_nbody(...) {
    ...
    deep_booster_alloc(MPI_COMM_WORLD, n, 1, &boosters);
    for (timestep=0; timestep<ntimesteps; timestep++) {
        remote = local;
        for (rank=0; rank<n; rank++) {
            #pragma omp task onto(boosters, my_rank) ...
            calculate_forces(...);
            #pragma omp task onto(boosters, my_rank) ...
            exchange_particles(...);
            remote = tmp;
        }
        #pragma omp task onto(boosters, my_rank) ...
        update_particles(...);
    }
    #pragma omp taskwait
    deep_booster_free(boosters);
}
```

Figure 9: Offloaded N-Body—*partial* version.

output data flushes to their *master* processes. As introduced earlier, due to the inherent overhead of task offloading, the OmpSs offload feature is intended to be used along with coarse grain task decomposition. Hence, the N-Body *global* offload implementation would be the preferred method to offload this computation to *booster* nodes, whereas the *partial* offload implementation is artificially developed with the purpose of presenting a bad-case scenario. In our experiments, we execute 10 time steps of the N-Body simulation.

Figure 10 shows the execution time of different configurations of the two N-Body implementations (maximum RSD: 0.9%). The interaction among 4 M particles is simulated using a *single booster*. The number after the `Error` key in the *partial* version states the task in which the error occurred, according to the order in which they appear in Figure 9. We follow the same naming convention in subsequent figures.

Figure 10 reveals that while in the `Global` case the protection mechanisms enabled by the use of the `recover` clause do not pose a noticeable performance impact, in `Partial` it introduces a 3.9% overhead. The `Error` cases include the overhead of MPI process respawning and data restoration, as well as that of the task reexecution. For example, `Error #1` includes the reexecution of the most computationally intensive task plus the overhead of using the `recovery` clause in

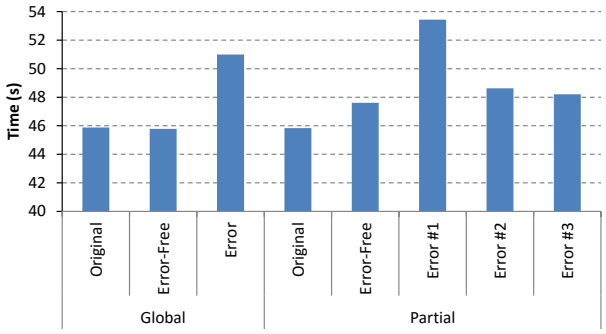


Figure 10: N-Body using one *booster* (4 M particles).

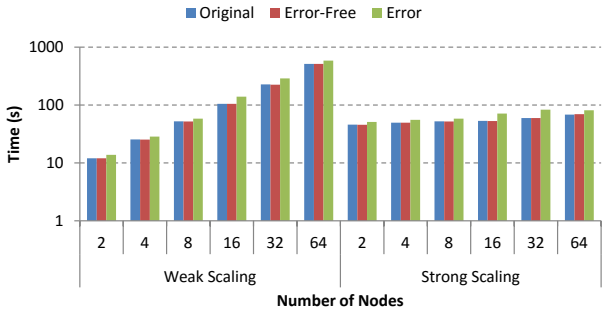


Figure 11: N-Body *global* scaling. Particles: weak–1 M/node; strong–4 M total.

this implementation (i.e., it is roughly `Partial Error-Free` plus `Global Error` minus `Global Error-Free`).

We next present a performance evaluation including both weak and strong scaling on up to 32 *master* plus 32 *booster* nodes. Figure 11 shows our results for the `global` implementation (maximum RSD: 2.2%). As we can see in the figure, the resilient offload feature does not impact scalability on this test case. We find the highest `Error-Free` overhead (2.5%) at the largest number of nodes in the strong scaling case. The recovery time varies between 11% and 39%, being higher for the large node counts in strong scaling due to the relative impact of the recovery overhead with respect to the computational load.

Experiencing more than a single fault in the less than 10 minutes these executions last is unlikely. If these faults occur at different points within the same group of offloaded tasks, the recovery process is the same as if a single failure happened. If, on the other hand, these are encountered during successive task executions, the application execution time may be easily inferred from the recovery times of the single-failure cases. Due to space constraints, we limit to present a study involving multiple successive failures for the FWI application in Section 6.4.

The scaling results for the *partial* implementation are depicted in Figure 12 (maximum RSD: 14.3%). As the figure shows, the overhead introduced by the offload fault-tolerance feature in this artificial test case is considerable, because of the reasons explained earlier in this section. While

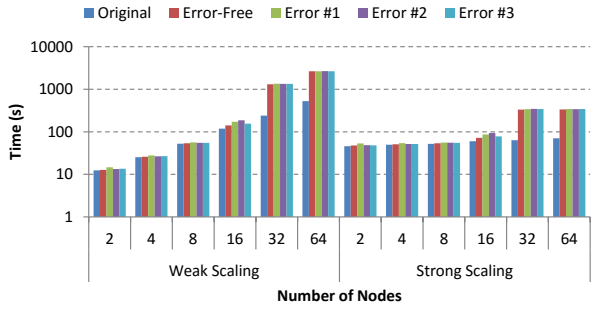


Figure 12: N-Body *partial* scaling. Particles: weak–1 M/node; strong–4 M total.

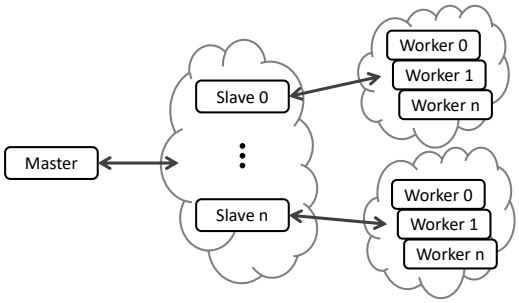


Figure 13: FWI offloaded structure.

`Error-Free` executions pose an overhead under 4% up to 8 nodes, this reaches almost 20% on 16 nodes, and over 400% for higher node counts. The recovery times are similar to those of the `global` case. As we mentioned earlier, this is a purposely-designed bad-case scenario and the OmpSs offload feature is not intended to execute fine-grained tasks.

6.4 Full Wave Inversion (FWI) Application

The Full Wave Inversion application analyzes the physical properties of the subsoil from seismic measurements. The area to be analyzed is divided in "shots", which are usually processed in a distributed fashion due to their size. Its structure using OmpSs offload semantics is depicted in Figure 13 and its implementation listed in pseudocode in Figure 14. An initial *master* process iterates over the different frequencies to explore. For each gradient, a number of shots are processed by a *kernel* function. Each of these is offloaded to a *slave* process. The *slaves*, after performing some preprocessing, split the computation of the kernel into several offloaded tasks (*workers*) and perform the required post-processing after waiting for their finalization. Several *test* post-processing *kernel* executions are required per *shot* after this process.

We configure our experiments to use 4 workers per slave, processing up to 1 GB of data each. We vary the number of shots from 2 to 16, using *up to 80 boosters* in different hosts. A single frequency, two gradients, and two tests are executed, leading to 6 `kernel` offloads to each *slave* during the execution. Nanos++ leverages the intranode parallelism using the 16 CPU cores of the compute nodes.

```
void kernel(...) {
    deep_booster_alloc(MPI_COMM_SELF, nw, 1, &workers);
    ...  // Preprocessing
    for (worker=0; worker<nw; worker++) {
        ...
        #pragma omp task onto(workers, worker) ...
        { ... }
    }
    #pragma omp taskwait
    ...  // Post-processing
    deep_booster_free(workers);
}

int main(void) {
    for (freq=0; freq<nfreqs; freq++) {  // Frequencies
        ...  // freqs require different number of shots
        deep_booster_alloc(MPI_COMM_WORLD, n, 1, &slaves);
        for (grad=0; grad<ngrads; grad++) {  // Gradients
            for (shot=0; shot<n; shot++) {
                #pragma omp task onto(slaves, shot) ...
                {
                    ...
                    kernel(...);
                }
            }
            #pragma omp taskwait
            for (test=0; test<ntest; test++) {
                for (shot=0; shot<n; shot++) {
                    #pragma omp task onto(slaves, shot) ...
                    {
                        ...
                        kernel(...);
                    }
                }
                #pragma omp taskwait
            }
        }
        deep_booster_free(slaves);
    }
}
```

Figure 14: FWI pseudocode.

Figure 15 shows the execution time of FWI in four different cases (the maximum RSD is 1.5%). In `Error Worker`, a failure appears in a *worker* process, whereas in `Error Slave`, it occurs in a *slave*. As we can see in the figure, the recovery feature does not noticeably impact the run time, being the maximum overhead a mere 0.56% (negative overheads just reflect variability and are indicative of the low performance impact). When a failure is detected, the task is reexecuted. The added execution time corresponds to the respawning procedure plus the reexecution of the tasks involved in the failure, including the necessary data restoring movements. Figure 16 represents an extrapolation of the execution time for different number of errors in the 64-worker case. A failure in a worker increases the execution time 18.3%, whereas if the failing process is a slave, the execution time is increased 20.3%. In the unlikely case of experiencing 5 process disconnection failures in the roughly 2.2 hours of execution, the run time would be approximately doubled.

7 CONCLUSION

Our experiments show no impact on the scalability properties of our test cases, revealing low runtime overhead and efficient recovery for coarse-grained tasks. The runtime overhead in our approach involves the synchronization of all sibling masters at the end of their offloaded tasks only in case these are collaborating (e.g., if they were not created using the

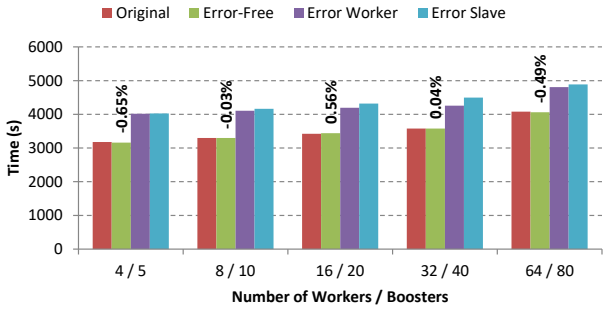


Figure 15: FWI execution time in 4 configurations.

Figure 16: FWI execution time extrapolation (64 workers).

`MPI_COMM_SELF` communicator). In that case, one would anyway expect some sort of user-level synchronization as well among the workers during the execution of their task, usually at least before finalization. Hence, targeting coarse-grained tasks [10], the runtime overhead introduced by our methodology should not be noticeable even in a larger number of nodes. In the event of a failure, the recovery time is basically dominated by the respawning latency, which should in any case improve upon restarting the entire application.

We hence have provided a use case demonstrating the viability of implementing effective and efficient fault tolerance using MPI-3 compliant techniques. Although we use OmpSs offload to showcase our idea, this approach should be applicable to other similar offload PMs such as [8].

ACKNOWLEDGMENTS

This research received funding from the European Community's 7th Framework Programme via the DEEP-ER project under Grant Agreement no. 610476. This work has also been supported by the Spanish Ministry of Science and Innovation (contract TIN2012-34557) and by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). Antonio J. Peña is cofinanced by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266. The authors thank Jorge Bellón, from BSC, for his technical support with the Nanos++ internals.

REFERENCES

[1] A. Barak and A. Shiloh. 2011. The MOSIX virtual OpenCL (VCL) cluster platform. In *Intel European Research and Innovation.*

[2] C. Clauss, T. Moschny, and N. Eicker. 2016. Dynamic process management with allocation-internal co-scheduling towards interactive supercomputing. In *Co-Scheduling of HPC Applications.*

[3] A. Amer et al. 2015. *MPICH User's Guide Version 3.2.* Argonne National Laboratory.

[4] A. J. Peña et al. 2014. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Comput.* 40, 10 (2014).

[5] A. Moody et al. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Int. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC).*

[6] A. Rezaei et al. 2014. Snapify: Capturing snapshots of offload applications on Xeon Phi manycore processors. In *Int. Symposium on High-Performance Parallel and Distributed Computing.*

[7] C. Cao et al. 2015. Design for a soft error resilient dynamic task-based runtime. In *International Parallel and Distributed Processing Symposium (IPDPS).*

[8] C. J. Newburn et al. 2013. Offload compiler runtime for the Intel® Xeon Phi coprocessor. In *International Parallel and Distributed Processing Symposium (IPDPS) Workshops.*

[9] D. G. Murray et al. 2011. CIEL: A universal execution engine for distributed data-flow computing. In *Symposium on Networked Systems Design and Implementation (NSDI).* 113–126.

[10] F. Sainz et al. 2015. Collective offload for heterogeneous clusters. In *Int. Conference on High Performance Computing (HiPC).*

[11] G. Bosilca et al. 2014. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience* 26, 17 (2014), 2772–2791.

[12] H. Takizawa et al. 2009. CheCUDA: A checkpoint/restart tool for CUDA applications. In *Parallel and Distributed Computing, Applications and Technologies.*

[13] H. Takizawa et al. 2011. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *International Parallel & Distributed Processing Symposium (IPDPS).*

[14] J. Wadden et al. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *International Symposium on Computer Architecture (ISCA).*

[15] K. S. Yim et al. 2011. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *Int. Conference on Parallel and Distributed Computing, Applications and Technologies.*

[16] L. Bautista-Gomez et al. 2011. FTI: High performance fault tolerance interface for hybrid systems. In *Int. Conference for High Perf. Computing, Networking, Storage and Analysis (SC).*

[17] M. Bougeret et al. 2014. Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications* 28, 2 (2014), 210–224.

[18] M. C. Kurt et al. 2014. Fault-tolerant dynamic task graph scheduling. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*

[19] O. Subasi et al. 2015. NanoCheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In *Parallel, Distributed and Network-Based Processing.*

[20] P. Balaji et al. 2010. PMI: A scalable parallel process-management interface for extreme-scale systems. In *EuroMPI.*

[21] S. Prabhakaran et al. 2015. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *International Parallel and Distributed Processing Symposium (IPDPS).*

[22] W. Bland et al. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254.

[23] W. Gropp and E. Lusk. 2004. Fault tolerance in message passing interface programs. *International Journal on High Performance Computing Applications* 18, 3 (2004), 363–372.

[24] L. Howes (Ed.). 2015. *The OpenCL Specification, Version 2.1.* Khronos OpenCL Working Group.

[25] NVIDIA 2016. *CUDA C Programming Guide 8.0.* NVIDIA.

[26] OpenACC-Standard.org. 2015. *The OpenACC© Application Programming Interface, Version 2.5.*

[27] OpenMP Architecture Review Board. 2015. *OpenMP Application Programming Interface* (4.5 ed.).

[28] A. J. Peña. 2013. *Virtualization of Accelerators in High Performance Clusters.* Ph.D. Dissertation. Universitat Jaume I.

[29] A. J. Peña, W. Bland, and P. Balaji. 2015. VOCL-FT: Introducing techniques for efficient soft error coprocessor recovery. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* IEEE/ACM.

[30] T. Warschko, J. M. Blum, and W. F. Tichy. 1998. ParaStation: Efficient parallel computing by clustering workstations. *Journal of Systems Architecture* 44, 3 (1998), 241–260.