Noname manuscript No.
(will be inserted by the editor)

Resilient MPI applications using an application-level checkpointing framework and ULFM

Nuria Losada · Iván Cores ·
María J. Martín · Patricia González

Received: 10 Sep 2015 / Accepted: 11 Jan 2016 – DOI: 10.1007/s11227-016-1629-7

Abstract Future exascale systems, formed by millions of cores, will present high failure rates, and long-running applications will need to make use of new fault tolerance techniques to ensure the successful execution completion. The Fault Tolerance Working Group, within the MPI forum, has presented the ULFM (User Level Failure Mitigation) proposal, providing new functionalities for the implementation of resilient MPI applications. In this work, the CPPC checkpointing framework is extended to exploit the new ULFM functionalities. The proposed solution transparently obtains resilient MPI applications by instrumenting the original application code. Besides, a multithreaded multilevel checkpointing, in which the checkpoint files are saved in different memory levels, improves the scalability of the solution. The experimental evaluation shows a low overhead when tolerating failures in one or several MPI processes.

Keywords Resilience · Checkpointing · Fault Tolerance · MPI

1 Introduction

Current petascale systems are formed by hundreds of thousands of cores. Schroeder and Gibson [14] have analysed failure data collected at two large high-performance computing sites, showing failure rates from 20 to more than 1,000 failures per year, depending mostly on system size. That can be translated in a failure every 8.7 hours. Future exascale systems will be formed by several millions of cores, and they will be hit by error/faults much more frequently than petascale systems due to their scale and complexity [6]. Therefore, long-running applications in these systems will need to use fault tolerance techniques to ensure the successful execution completion.

Nuria Losada · Iván Cores · María J. Martín · Patricia González
Grupo de Arquitectura de Computadores, Universidade da Coruña, Spain
Tel.: +34 881011212
Fax: +34 981167160
E-mail: { nuria.losada, ivan.coresg, mariam, patricia.gonzalez }@udc.es

The MPI (Message Passing Interface) standard is the most popular parallel programming model in petascale systems. However, MPI lacks fault tolerance support. By default, the entire MPI application is aborted upon a single process failure. Besides, the state of MPI will be undefined upon failure and, thus, there are no guarantees that the MPI program can successfully continue its execution. Thus, traditional fault tolerant solutions for MPI applications rely on checkpointing, relaunching a new MPI job for restarting the application. However, a complete restart is unnecessary, since most of the nodes will still be alive. Moreover, a complete restart introduces overheads both for re-queuing the MPI job and for moving the checkpointed data across the cluster to the new granted resources.

In the last years new methods have emerged to provide fault tolerance to MPI applications, such as failure avoidance approaches [7, 16] that preemptively migrate processes from processors that are about to fail. Unfortunately, these solutions are not able to cope with already happened failures. Recently, the Fault Tolerance Working Group within the MPI forum proposed the ULFM (User Level Failure Mitigation) interface [4] to integrate resilience capabilities in the future MPI 4.0. It includes new semantics for process failure detection, and communicator revocation and reconfiguration. Thus, it enables the implementation of resilient MPI applications, that is, applications that are able to recover themselves from failures. Nevertheless, incorporating the ULFM capabilities in already existing codes is a complex and time-consuming task.

In this work, the checkpointing tool CPPC [12] is extended to use the new functionalities provided by ULFM to transparently obtain resilient MPI applications from generic MPI SPMD (Single Program Multiple Data) programs. proposed solution is able to detect failures in one or more processes, and to recover from them, without re-queuing nor stopping the MPI job. Besides, a multithreaded multilevel checkpointing is implemented, storing copies of the checkpoint files in different memory levels. This technique reduces the overhead and network contention upon failure, as less data have to be moved across the cluster, while the cost of checkpointing is not increased.

This paper is structured as follows. Section 2 introduces the CPPC framework, while Section 3 details its extension to obtain resilient MPI applications. The multithreaded multilevel checkpointing technique is described in Section 4 and the experimental evaluation is presented in Section 5. Section 6 covers the related work. Finally, Section 7 concludes this paper.

2 CPPC overview

CPPC [12] is an open-source checkpointing tool for MPI applications available under GPL license at `http://cppc.des.udc.es`. It is implemented at the application level, and, thus, it is independent of the operating system and any higher-level framework used.

CPPC appears to the final user as a compiler tool and a runtime library. At compile time the CPPC source-to-source compiler automatically transforms a code into an equivalent fault-tolerant version by adding calls to the CPPC library. The resulting fault tolerant code can be seen in Fig. 1. Instrumentation is added to perform the following actions:

```
int error;
int GLOBAL_COMM;

void function1(){
   […]
   MPI_…(MPI_COMM_WORLD, …);
   function2();/*MPI calls inside*/
   […]
}
void function2(){ … }

int main(){
  CPPC_Init_configuration();
  MPI_Init()
  CPPC_Init_state();

  MPI_Comm_split(MPI_COMM_WORLD, …, NEW_COMM);
  if (CPPC_Jump_next()) goto CPPC_REC_1;
  […]
```

```
  CPPC_REC_1:
  <CPPC_Register() block>
  if (CPPC_Jump_next()) goto CPPC_REC_2;
  […]

  for(i=0;i<niters;i++){
    CPPC_REC_2:
    CPPC_Do_Checkpoint();

    […]
    MPI_…(MPI_COMM_WORLD …);

    function1();/*MPI calls inside*/
  }
  […]
  <CPPC_Unregister() block>
  CPPC_Shutdown();
}
```

Fig. 1 CPPC instrumentation for stop and restart fault-tolerant applications.

- **Configuration and initialization:** at the beginning of the application the routines `CPPC_Init_configuration()` and `CPPC_Init_state()` configure and initialize the necessary data structures for the library management.
- **Registration of variables:** the routine `CPPC_Register()` explicitly marks for their inclusion in checkpoint files those variables identified by a liveness analysis [8] as necessary for the successful recovery of the application. During restart, this routine also recovers the values from the checkpoint files to their proper memory location.
- **Checkpoint:** the `CPPC_Do_checkpoint()` routine dumps the checkpoint file. A checkpoint file will be generated every N calls to this function, being N user-defined. At restart time this routine checks restart completion.
- **Shutdown:** the `CPPC_Shutdown()` routine is added at the end of the application to ensure the consistent system shutdown.

Checkpoint file sizes are reduced by using the liveness analysis and the zero-blocks exclusion technique [8], which avoids the storage of memory blocks that contain only zeros. Moreover, a multithreaded dumping performs a copy in memory of the checkpointed data so that it can be dumped to disk by an auxiliary thread, overlapping the checkpoint file writing with the computation of the application.

The state files generated by CPPC are portable, allowing the restart on different architectures and/or operating systems. Portability is achieved by using HDF5, a portable storage format, and by avoiding the inclusion in the checkpoint files of architecture-dependent state. This state is recovered through the re-execution of the code responsible for creating such state in the original execution. The compiler automatically identifies both the variables to be checkpointed and the non-portable code to be re-executed upon restart.

As for checkpoint consistency, the basic difference between sequential and parallel applications is the existence of dependencies imposed by inter-process communications. The CPPC compiler performs a static analysis of inter-process communication and automatically identifies safe points, code locations where it is guaranteed that there are no in-transit, nor inconsistent messages, avoiding the domino effect. Besides, a heuristic identifies the most computationally expensive loops and inserts a checkpoint function in the first safe point of these loops. By statically ensuring that checkpoints may occur only at selected safe points, no inter-process communications or runtime synchronizations are necessary when checkpointing.

During restart, the application processes perform a negotiation phase to identify the most recent valid recovery line, formed by the newest checkpoint file available simultaneously to all processes. The restart phase has two fundamental parts: reading the checkpoint data into memory and recovering the application state. The first step is encapsulated inside the routine `CPPC_Init_state()`. However, the actual reconstruction of the application state is achieved through the ordered execution of certain blocks of code called RECs (Required-Execution Code): the configuration and initialization block, variable registration blocks, checkpoint blocks, and non-portable state recovery blocks, such as the creation of communicators. The compiler inserts control flow code (labels and conditional jumps) to ensure an ordered re-execution.

3 Resilient MPI applications with CPPC

This section describes how CPPC is extended to exploit the new ULFM functionalities to transparently obtain resilient applications from general MPI SPMD codes. To maximize the applicability, the proposed approach is non-shrinking and it uses a global backward recovery based on checkpointing:

– Non-shrinking recovery: the number of executing processes is preserved after a failure. MPI SPMD applications generally base their distribution of data and computation on the number of running processes. Thus, shrinking solutions are restricted to applications which tolerate a redistribution of data and workload among the processes during runtime.
– Backward recovery based on checkpoint: upon failure the application is restarted from a previous saved state. Forward recovery solutions attempt to find a new state to successfully continue the execution of the application. Unfortunately, forward recovery solutions are application-dependent, and, thus, unsuitable to be applied in a general approach.
– Global recovery: the application repairs a global state to survive the failure. In MPI SPMD applications that means restoring the state of all application processes to a saved state, in order to obtain the necessary global consistency to resume the execution. Local recovery solutions attempt to repair failures by restoring a small part of the application, e.g., a single process. However, due to interprocess communication dependencies, these solutions require the use of message logging techniques for its general application.

Next subsections describe the strategy upon failure: all the survivor processes need to detect the failure, so that the global communicator is reconfigured and a global consistent application state is recovered, allowing the execution to be resumed. Fig. 2 illustrates the whole procedure and Fig. 3 shows an example of the new CPPC instrumentation added to carry it out.

3.1 Failure detection

By default, when a process fails, the MPI application is aborted. The routine `MPI_Comm_set_errhandler()` is used to set `MPI_ERRORS_RETURN` as the default er-

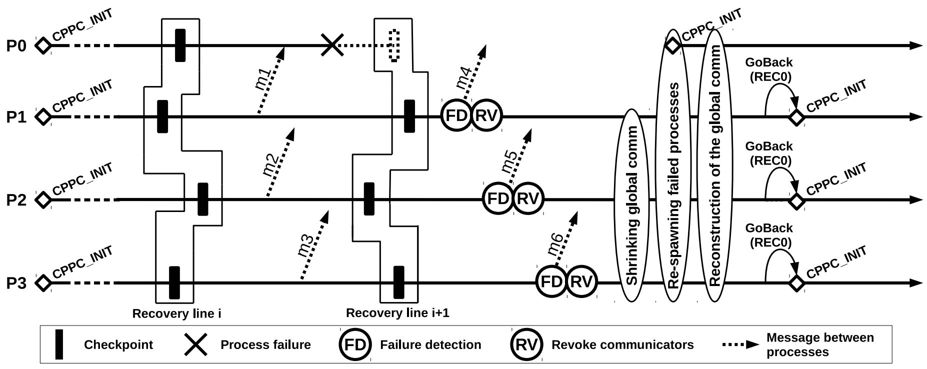

Fig. 2 Obtaining resilient MPI applications with CPPC and ULFM.

ror handler on each communicator, so that ULFM defined error codes are returned after a failure. Each MPI function call is instrumented with a call to the `CPPC_Check_errors()` routine to check whether the returned value corresponds with a failure, as shown in Fig. 3. Within the `CPPC_Check_errors()` routine, whose pseudocode is shown in Fig. 4, the survivor processes detect failures and trigger the recovery process.

However, the failure is only detected locally by those survivor processes involved in communications with failed ones. To reach a global detection, the failure has to be propagated to all other survivors by revoking all the communicators used by the application. CPPC keeps a reference to the new communicators created by the application by means of the `CPPC_Register_comm()` function. When a failure is locally detected, the ULFM `MPI_Comm_revoke()` function is invoked over the global (`MPI_COMM_WORLD`) and the registered communicators, assuring a global failure detection. In the example shown in Fig. 2, process $P1$ detects that $P0$ failed because they are involved in a communication, while processes $P2$ and $P3$ detect the failure after the communicators are revoked by $P1$.

Lastly, in order to guarantee failure detection in reasonable time, the function `MPI_Iprobe()` is invoked within the `CPPC_Do_checkpoint()` routine. This avoids excessive delays in failure detection in absence of communications.

3.2 Reconfiguration of the MPI global communicator

As commented previously, this proposal preserves the total number of processes of the application. Besides, in the SPMD model, the processes ranks distinguish the roles of the processes in the execution so they must be preserved as well after a failure. Thus, the failed processes need to be re-spawned and their ranks need to be restored.

Those communicators created by the application, which derive from the global communicator (`MPI_COMM_WORLD`), will be reconstructed by re-executing the MPI calls used for creating them in the original execution. On the other hand, the global communicator has to be reconfigured after failure detection. Firstly, the failed processes are excluded from the global communicator using the ULFM

```
int error;
int GLOBAL_COMM;

void function1(){
  […]
  error = MPI_…(GLOBAL_COMM …);
  if(CPPC_Check_errors(error)) return;
  function2();/*MPI calls inside*/
  if(CPPC_Go_init()) return;
  […]
}
void function2(){ … }

int main(){
  CPPC_GOBACK_REC_0:
  CPPC_Init_configuration();
  if(!CPPC_Go_init()) MPI_Init()
  CPPC_Init_state();

  GLOCAL_COMM=CPPC_Get_comm();
  error = MPI_Comm_split(GLOBAL_COMM, …, NEW_COMM);
  if(CPPC_Check_errors(error))goto CPPC_GOBACK_REC_0;
  CPPC_Register_comm(NEW_COMM)
  if (CPPC_Jump_next()) goto CPPC_REC_1;
  […]
```

```
  CPPC_REC_1:
  <CPPC_Register() block>
  if (CPPC_Jump_next()) goto CPPC_REC_2;
  […]

  for(i=0;i<niters;i++){
    CPPC_REC_2:
    CPPC_Do_Checkpoint();
    if(CPPC_Go_init()) goto CPPC_GOBACK_REC_0;
    […]
    error = MPI_…(GLOBAL_COMM …);
    if(CPPC_Check_errors(error))
       goto CPPC_GOBACK_REC_0;

    function1();/*MPI calls inside*/
    if(CPPC_Go_init()) goto CPPC_GOBACK_REC_0;
  }
  […]
  <CPPC_Unregister() block>
  CPPC_Shutdown();
}
```

Fig. 3 CPPC instrumentation for resilient MPI applications.

```
bool CPPC_Check_errors(int error_code){
 if( error_code == process failure ){      ⎫
   //Revoke communicators                  ⎬ Failure
   For c in application_communicators      ⎬ detection
     MPI_Comm_revoke(…, c, …)              ⎭

   //Shrink global communicator            ⎫
   MPI_Comm_shrink(…)                      ⎪
   //Re-spawn failed processes             ⎪
   MPI_Comm_spawn_multiple(…)              ⎪ Reconfiguration
   //Reconstruct communicator              ⎬ of the global
   MPI_Intercomm_merge(…)                  ⎪ communicator
   MPI_Comm_group(…)                       ⎪
   MPI_Group_incl(…)                       ⎪
   MPI_Comm_create(…)                      ⎭

   //Start the recovery process            ⎫ Start the
   return true;                            ⎭ recovery
 }
 //No error detected
 return false;
}
```

Fig. 4 `CPPC_Check_errors()` pseudocode.

function `MPI_Comm_shrink()` and they are re-spawned using the MPI function `MPI_Comm_spawn_multiple()`. Then, the dynamic communicator management facilities provided in MPI-2 are used to reconstruct the global communicator, so that, after a failure, the survivor processes keep their original ranks, while each one of the re-spawned ones takes over a failed process. To ensure that the correct global communicator is used, the application obtains it by means of the new `CPPC_Get_comm()` function, and stores it in a global variable, as shown in Fig. 3. The application uses this global variable instead of the named constant `MPI_COMM_WORLD`, allowing CPPC to transparently replace it with the reconfigured communicator after a failure.

3.3 Recovery of the application

A crucial step in ensuring the success of the restart is to be able to conduct the application to a consistent global state before resuming the execution. Solutions based on checkpoint, like this proposal, recover the application state rolling back to the most recent valid recovery line. However, CPPC only dumps to checkpoint files portable state, while non-portable state is recovered through the ordered re-execution of certain blocks of code (RECs). Therefore, to achieve a consistent global state, all processes must go back to the beginning of the application code so that they can re-execute the necessary RECs (including those for the creation of the derived communicators). The re-spawned processes are already in this function after the reconfiguration of the communicators. However, survivor processes are located at the point of the application code where they have detected the failure and they need to go back in the application control flow. This is performed by reversed conditional jumps introduced in two instrumentation blocks: the `CPPC_Check_errors()` and the `CPPC_Go_init()` blocks. While the `CPPC_Check_errors()` blocks are placed after MPI calls, the `CPPC_Go_init()` blocks are located after those application functions containing MPI calls. As shown in Fig. 3, the reversed conditional jumps consist in a `goto` with a label or a `return` instruction, depending if it is placed in the main program or in an internal function.

Once all the processes reach the beginning of the application code, an usual CPPC restart is performed. First, processes negotiate the most recent valid recovery line. In the example shown in Fig. 2, all processes will negotiate to recover the application state from recovery line i. Then, the negotiated checkpoint files containing portable state are read and the actual reconstruction of the application state is achieved through the ordered re-execution of RECs, recovering also the non-portable state. Finally, the application execution continues normally when the point where the checkpoint files were generated is reached.

4 Improving scalability: multithreaded multilevel checkpointing

CPPC uses a multithreaded dumping to minimize the overhead introduced when checkpointing. The multithreaded dumping overlaps the checkpoint file writing with the computation of the application. When a MPI process determines that a checkpoint file must be generated (according to the checkpointing frequency), it prepares the checkpointed data, performs a copy in memory of that data, and creates an auxiliary thread that builds and dumps to disk the checkpoint file in background, while the application processes continue their execution.

With the extension of CPPC to exploit the ULFM functionalities, a multilevel checkpointing is implemented to minimize the amount of data to be moved across the cluster, thus, reducing the recovery overhead when failures arise and increasing the scalability of the proposal. As in a memory hierarchy, in which higher levels present lower access time but less capacity and larger miss rates, this technique stores the checkpoint files in three different locations:

– Memory of the compute node: each process maintains the copy in memory of the last checkpoint file generated until a new checkpoint file is built.
– Local disk: processes also save their checkpoint files in local storage.
– Remote disk: all the checkpoint files generated by the application are stored in an NFS mounted directory in a remote disk.

The multilevel checkpointing is performed in background by the auxiliary threads, thus, the cost of checkpointing is not increased. During recovery, the application processes perform a negotiation phase to identify the most recent valid recovery line, formed by the newest checkpoint file available simultaneously to all processes. When using the multilevel technique the three checkpointing levels are inspected during the negotiation, and the application processes choose the closest copy of the negotiated checkpoint files.

5 Experimental evaluation

The application testbed used is comprised of three benchmarks with different checkpoint file sizes and communication patterns. The ASC Sequoia Benchmark SPhot [2] is a physics package that implements a Monte Carlo Scalar PHOTon transport code. It was run using 6144 as the NRUNS parameter. The Himeno benchmark [10] is a Poisson equation solver using the Jacobi iteration method. It was run fixing NN to 24000 and using 512x256x256 as grid size. Finally, MOCFE-Bone [17] simulates the main procedures in a 3D method of characteristics (MOC) code for numerical solution of the steady state neutron transport equation. MOCFE-Bone was run using 4 energy groups, 8 angles, a mesh of 19^3 doing strong scaling in space, and a trajectory spacing of $0.5cm^2$. The CPPC version used was 0.8.1, working along with HDF5 v1.8.11 and GCC v4.4.7. The ULFM commit icldistcomp-ulfm-67d6cc5b9cfa was used with the default configuration parameters and the agreement algorithm number 1. Finally, the Portable Hardware Locality (hwloc) [5] is used for the binding of the processes to the CPU cores.

Each node of the cluster used for the experiments consists of two Intel Xeon E5-2660 Sandy Bridge-EP 2.2 GHz processors with Hyper-Threading support, with 8 cores per processor and 64 GB of RAM, interconnected to an InfiniBand FDR and a Gigabit Ethernet networks. The experiments are run spawning 16 MPI process per node, one per core. When checkpointing, each MPI process will create an auxiliary thread to dump data to disk.

5.1 Instrumentation and checkpointing overhead

The instrumentation overhead is measured in the execution of the CPPC instrumented applications without generating any checkpoint files. On the other hand, the checkpointing runtime corresponds with the execution in which only one checkpoint is taken when the 50% of the computation has completed and using the multithreaded multilevel checkpointing technique explained in Section 4. Fig. 5 shows the original, the instrumentation, and the checkpointing runtimes varying the number of processes. The total state saved to disk in each application (the addition of the checkpoint files generated by each process) is also represented in the figure. For SPhot the checkpoint file of each individual process is constant, no matter how many processes run the application, thus, the total checkpoint file size increases with the number of processes. On the other hand, for Himeno and MOCFE-Bone, the application data is distributed among the processes, therefore, each individual checkpoint file size decreases as more processes run the application, and the total checkpoint file size remains almost constant.

The instrumentation overhead is always very low, below 1.7 seconds. As regards the checkpointing, the maximum overhead is 7.9 seconds. The two main

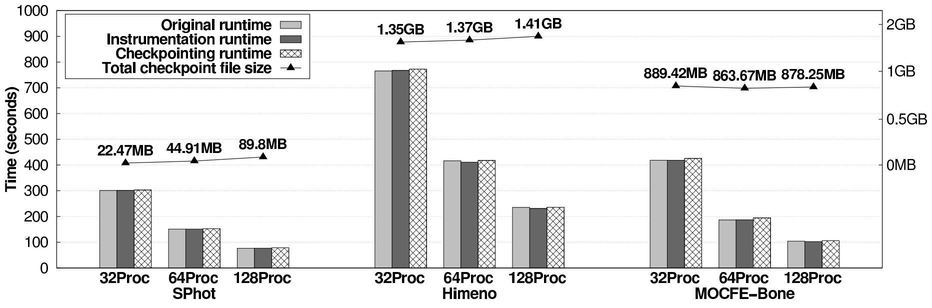

Fig. 5 Runtimes for the testbed benchmarks, specifying number of processes and total checkpoint file size.

sources of overhead in the checkpointing operation are: the copy in memory of the checkpointed data (including the application of the CPPC zero-blocks exclusion technique), and the dumping to disk. Thanks to the use of the multithreaded checkpointing technique, the overhead of dumping the checkpoint files to disk is significantly reduced.

5.2 Resilient MPI applications

The performance of the proposal is evaluated inserting one-process or full-node failures by killing one or sixteen MPI processes, respectively. Failures are introduced when the 75% of the application has completed and the applications are recovered using the checkpoint files generated at the 50% of the execution. In all the experiments, the survivor processes recover from the copy in memory of the checkpoint files. When one process fails, it is re-spawned in the same compute node, thus, it uses the checkpoint file in local storage. When a node fails, the failed processes are re-spawned in a different compute node: an already in use node, overloading it; or an spare node, pre-allocated for this purpose when scheduling the MPI job. In both cases, the failed processes use the checkpoint files located in the remote disk.

The analysis of the overhead introduced by the proposal requires the study of the different operations it involves. Fig. 6 shows the times, in seconds, of each operation performed to obtain resilient MPI applications, indicating whether one-process or full-node failures are introduced, and for the late one, if a compute node is overloaded or an spare one is used for the recovery. Failure detection times are deeply dependent on the application, as they depend on the frequency of MPI calls. Table 1 shows, for each application, the average number of MPI calls per second in the process that performs the fewest MPI calls. SPhot is the application with the smallest number and it also presents the largest detection time. The communicator revocation times are low and they slightly increase with the number of processes. The same tendency can be observed in the reconfiguration operations: the communicator shrinking, the process spawning and the communicator reconstruction. Even though the processes spawning times remain low, they increase with the number of failed processes. Furthermore, spawning times are larger when

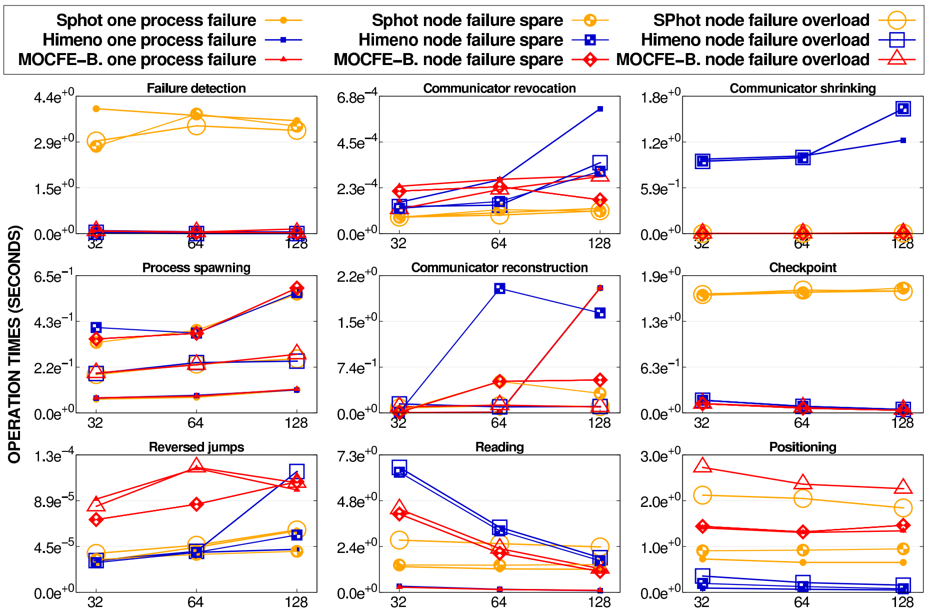

Fig. 6 Operations to obtain resilience.

	# MPI CALLS PER SECOND DONE BY THE PROCESS WITH FEWEST CALLS		
	32PROCS.	64PROCS.	128PROCS.
SPhot	1.36	1.43	1.57
Himeno	815.18	1498.38	2652.24
MOCFE-Bone	1352.21	3034.68	5431.90

Table 1 Number of calls to the MPI library per second performed by the process that less calls does, which determines the detection time.

a spare node is used than when overloading a node, assumingly because a new ORTE daemon has to be launched in the spare node.

With regard to the CPPC operations, the time spent in the reversed conditional jumps for the recovery upon failure is negligible in all cases. The checkpoint, reading, and positioning times are consistent with the state registered by each process for its inclusion in the checkpoint files. Note that, the size of the registered state can be different from the size of the generated checkpoint file, as CPPC applies the zero-blocks exclusion technique [8], which avoids the dumping to disk of the memory blocks containing only zeros. During the checkpoint operation, all the registered data is inspected to identify zero-blocks. Also, during the reading and positioning operations, zero-blocks are reconstruct and moved to their proper memory location, respectively. Thus, checkpoint, reading, and positioning are influenced by the size of registered data. Table 2 shows, for each application, the average size of state registered by each process and the average checkpoint file size that each process generates after applying the zero-blocks exclusion technique. For instance, in SPhot even though the total checkpoint file size is inferior to 100 MB, the total registered state corresponds with several gigabytes before the application

	AVERAGE SIZE PER PROCESS (MB): REGISTERED DATA → CKPT FILE EXCLUDING ZERO-BLOCKS		
	32 PROCESSES	64 PROCESSES	128 PROCESSES
SPhot	586.23 → 0.70	586.23 → 0.70	586.23 → 0.70
Himeno	61.49 → 43.25	31.49 → 21.97	16.15 → 11.32
MOCFE-Bone	44.90 → 27.79	22.89 → 13.49	12.30 → 6.86

Table 2 Average size (MB) of data registered by each process (including zero-blocks) and average size (MB) of the checkpoint file generated by each process (excluding zero-blocks).

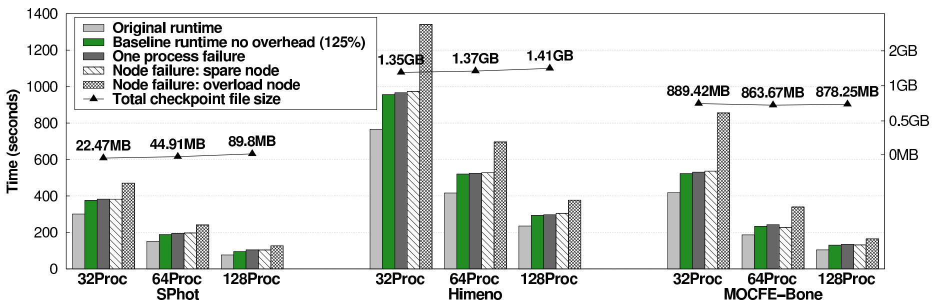

Fig. 7 Runtimes when introducing failures varying the number of processes. The baseline runtime in which no overhead is introduced is included for comparison purposes.

of the zero-blocks exclusion technique. Therefore, the CPPC operations are more costly for SPhot when comparing with applications that generate larger checkpoint files. Besides, due to the use of the multilevel checkpointing, the reading phase depends on the location of the negotiated checkpoint files. Consequently, the reading times for the experiments introducing one process failures present the lowest reading time, as the failed process reads the checkpoint file from the local storage of the compute node. On the other hand, in the experiments introducing node failures, the reading operation presents a higher cost because the failed processes use the copy of the checkpoint files in the remote disk.

Finally, the total overhead introduced by the proposal is studied. The failure-introduced runtimes are measured, including the execution until the failure, the detection and recovery from it, and the completion of the execution afterwards. As checkpointing takes place at 50% of the execution and the failure is introduced at 75%, the minimum runtime of an execution tolerating a failure will be 125% of the original runtime (75% until failure plus 50% from the recovery point until the end of the execution). Therefore, we consider this time as the baseline runtime, and the overhead of the proposal is measured as the difference between the baseline and the testbed failure-introduced runtimes.

Fig. 7 shows the original runtimes, as well as the baseline and the testbed failure-introduced runtimes. In the experiments, when only one MPI process is killed, the total cost of tolerating the failure is, on average, 6.6 seconds, introducing 3.6% of relative overhead with respect to the original runtimes. Similarly, when a full-node failure is recovered using a spare node, the absolute overhead is 8.7 seconds on average, introducing 3.7% of relative overhead. However, when an already in use node is overloaded, runtimes are larger, as both the computation after the failure and the operations to recover the applications are slower.

Regarding the relation between the number of running processes and the total overhead, we observed that there are not significant differences for the experiments introducing one process failures or node failures using a spare node. However, in the experiments overloading a computation node, the total overhead decreases with the number of processes. As more processes execute the applications, less work corresponds to each one of them, and the impact of the overloading is reduced.

6 Related work

Some works in the literature [3, 9] have proposed extensions to the MPI interface to cope with failures in MPI processes and obtain resilience. However, none of this proposals were included in the MPI standard so far.

Recently, different approaches for resilience using the new ULFM functionalities have emerged. Some of these solutions are Algorithm-Based Fault Tolerance (ABFT) techniques, which means that they are specific to one or a set of applications and they can not be generally applied, such as the proposals of Laguna et al. [11] and Ali et al. [1]. In contrast, the works of Sato et al. [13] and Teranishi and Heroux [15], like the one in this proposal, focus on obtaining general MPI resilient applications. However, both of them rely on the developers to instrument their MPI applications in order to obtain fault tolerance support, which is, in general, a complex and time-consuming task.

7 Concluding remarks

In this proposal the CPPC checkpointing tool is extended to exploit the new ULFM functionalities to transparently obtain resilient applications from general MPI SPMD programs. By means of the CPPC instrumentation of the original application code, failures in one or several MPI processes are tolerated using a non-shrinking backwards recovery based on checkpointing. Besides, a multithreaded multilevel checkpointing stores a copy of the checkpoint files in different levels of memory, minimizing the amount of data to be moved upon failure, and thus, reducing the recovery overhead without increasing the checkpointing overhead.

The experimental evaluation analyzes the behaviour when one or sixteen MPI processes (a full compute node) fail. Furthermore, in case of a node failure, two different scenarios are considered: the failed processes are re-spawned in a spare node or in an already in use one (overloading it). Results show the low overhead of the solution.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P, CAPAP-H5 network TIN2014-53522-REDT, and the predoctoral grant of Nuria Losada ref. BES-2014-068066) and by the Galician Government (Xunta de Galicia) and FEDER funds of the EU under the Consolidation Program of Competitive Research (ref. GRC2013/055).

References

1. Ali, M., Southern, J., Strazdins, P., Harding, B.: Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver. In: IEEE International Parallel Distributed Processing Symposium Workshops, pp. 1169–1178 (2014)
2. ASC Sequoia Benchmark Codes: https://asc.llnl.gov/sequoia/benchmarks/. Last accessed: September 2015
3. Aulwes, R., Daniel, D., Desai, N., Graham, R., Risinger, L., Taylor, M.A., Woodall, T., Sukalski, M.: Architecture of LA-MPI, a network-fault-tolerant MPI. In: International Parallel and Distributed Processing Symposium, pp. 15– (2004)
4. Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.: An evaluation of user-level failure mitigation support in MPI. In: Recent Advances in the Message Passing Interface, *Lecture Notes in Computer Science*, vol. 7490, pp. 193–203. Springer (2012)
5. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: Euromicro International Conference on Parallel, Distributed and Network-Based Computing. Pisa, Italy (2010)
6. Cappello, F.: Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. International Journal of High Performance Computing Applications **23**(3), 212–226 (2009)
7. Cores, I., Rodríguez, G., González, P., Martín, M.: Failure avoidance in MPI applications using an application-level approach. The Computer Journal **57**(1), 100–114 (2014)
8. Cores, I., Rodríguez, G., Martín, M., González, P., Osorio, R.: Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes. New Generation Computing **31**(3), 163–185 (2013)
9. Fagg, G., Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 1908, pp. 346–353. Springer (2000)
10. Himeno Benchmark: http://accc.riken.jp/2444.htm. Last accessed: September 2015
11. Laguna, I., Richards, D., Gamblin, T., Schulz, M., de Supinski, B.: Evaluating User-Level Fault Tolerance for MPI Applications. In: European MPI Users' Group Meeting, EuroMPI/ASIA '14, pp. 57–62 (2014)
12. Rodríguez, G., Martín, M., González, P., Touriño, J., Doallo, R.: CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. Concurrency and Computation: Practice and Experience **22**(6), 749–766 (2010)
13. Sato, K., Moody, A., Mohror, K., Gamblin, T., De Supinski, B., Maruyama, N., Matsuoka, S.: FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1225–1234 (2014)
14. Schroeder, B., Gibson, G.: A Large-Scale Study of Failures in High-Performance Computing Systems. IEEE Transactions on Dependable and Secure Computing **7**(4), 337–350 (2010)
15. Teranishi, K., Heroux, M.: Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In: European MPI Users' Group Meeting, pp. 51–56 (2014)
16. Wang, C., Mueller, F., Engelmann, C., Scott, S.: Proactive process-level live migration in HPC environments. In: ACM/IEEE conference on Supercomputing, pp. 1–12 (2008)
17. Wolters, E., Smith, M.: MOCFE-Bone: the 3D MOC mini-application for exascale research. Tech. rep., Argonne National Laboratory (ANL) (2013)