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

Assessing Resilient versus Stop-and-Restart Fault-Tolerant Solutions in MPI Applications

Nuria Losada · María J. Martín · Patricia González

DOI: 10.1007/s11227-016-1863-z

Abstract The MPI (Message Passing Interface) standard is the most popular parallel programming model for distributed systems. However, it lacks fault tolerance support and, traditionally, failures are addressed with stop-and-restart checkpointing solutions. The proposal of ULFM (User Level Failure Mitigation) for the inclusion of resilience capabilities in the MPI standard provides new opportunities in this field, allowing the implementation of resilient MPI applications, i.e. applications that are able to detect and react to failures without stopping their execution. This work compares the performance of a traditional stop-and-restart checkpointing solution with its equivalent resilience proposal. Both approaches are built on top of CPPC (ComPiler for Portable Checkpoiting), an application-level checkpointing tool for MPI applications, and they allow to transparently obtain fault-tolerant MPI applications from generic MPI SPMD (Single Program Multiple Data) programs. The evaluation is focussed on the scalability of the two solutions, comparing both proposals using up to 3072 cores.

Keywords Resilience · Checkpointing · Fault Tolerance · MPI

1 Introduction

Di Martino et al. [8] have studied during 518 days the Cray supercomputer Blue Waters, reporting that 1.53% of applications running on the machine failed because of system-related issues. This means that, on average, a failure arises every 15 minutes. Future exascale systems will be formed by several millions of cores, and they will be hit by error/faults much more frequently due to their scale and complexity. Therefore, long-running applications will need

Nuria Losada · María J. Martín · Patricia González
Grupo de Arquitectura de Computadores, Universidade da Coruña, Spain
E-mail: { nuria.losada, mariam, patricia.gonzalez }@udc.es

to use fault tolerance techniques to ensure the completion of their execution in these systems and to save energy.

The MPI (Message Passing Interface) standard is the most popular parallel programming model in distributed memory 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 in the event of a failure and there are no guarantees that the MPI program can successfully continue its execution. Thus, traditional fault tolerant solutions for MPI applications rely on stop-and-restart checkpointing: the computation state is periodically saved to stable storage into checkpoint files, so that, when a failure occurs, the application can be relaunched and its state recovered. However, when a failure arises it frequently has a limited impact and affects only a subset of the cores or computation nodes in which the application is being run. Thus, most of the nodes will still be alive. In this context, aborting the MPI application to relaunch it again introduces unnecessary recovery overheads and more efficient solutions need to be explored.

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 standard. 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.

CPPC (ComPiler for Portable Checkpointing) [19] is an open-source checkpointing tool for MPI applications, which originally applies a stop-and-restart checkpointing strategy. CPPC has been extended to exploit the new resilience capabilities provided by ULFM [14]. The proposal is able to detect failures in one or multiple processes, and to recover from them, without stopping the execution of the application and preserving the number of MPI processes running the application. This proposal transparently obtains resilient MPI applications from generic MPI SPMD (Single Program Multiple Data) programs by automatically instrumenting the original application code. In this work, the CPPC resilience proposal is exhaustively evaluated in a large machine, considering different fault scenarios, to analyze the scalability of the proposal and to detect possible bottlenecks. Besides, it is compared with the traditional stop-and-restart checkpointing strategy in terms of performance and benefits.

The resilience solution based on CPPC is, to the best of our knowledge, the only current proposal that automatically and transparently transforms MPI codes into resilience ones, which constitutes an appealing feature to existing HPC applications. The thorough evaluation presented here, carried out with large-scale applications on a large system, will be useful to ongoing researchs on resilience solutions towards exascale computing.

This paper is structured as follows. Section 2 covers the related work. Section 3 introduces the CPPC framework, describing the traditional stop-and-restart approach, as well as the resilience proposal. The experimental evaluation is presented in Section 4. Finally, Section 5 concludes this paper.

2 Related work

In line with previous works [3,9,11], the ULFM interface [4] is the last effort to include fault tolerance capabilities in the MPI standard. It is a low-level API that supports a variety of fault tolerance models and thus, it is responsibility of the user to design the recovery strategy.

In the literature, there exist different proposals to implement resilient applications using ULFM, most of them specific to one or a set of applications [16, 5,12,1,18]. Bland et al. [5] and Pauli et al. [16] focused on Monte Carlo methods. Laguna et al. evaluate ULFM on a massively scalar molecular dynamics code [12]. Partial Differential Equation (PDE) codes are targeted by Ali et al. [1] and by Rizzi et al. [18]. All these proposals take into account the particular characteristics of the applications to simplify the recovery process. A customized solution allows reducing the recovery overhead upon failure, e.g., simplifying the detection of failures by checking the status of the execution in specific points; avoiding the re-spawning of the failed processes when the algorithm tolerates shrinking the number of the MPI processes; or recovering the application data by means of its properties as an alternative to checkpointing. In contrast, unlike our proposal, they can not be generally applied to any SPMD application.

Other alternatives to ULFM to build resilient applications are Reinit [13], FMI [20] or NR-MPI [21]. In contrast with ULFM, which proposes a low-level API that supports a variety of fault tolerance models, these alternatives propose a simplified interface towards a non-shrinking model, repairing the MPI inner state upon failure, and re-spawning the failed processes. Reinit proposes a prototype fault-tolerance interface for MPI, suitable for global, backward, non-shrinking recovery. FMI is a prototype programming model with a similar semantic to MPI that handles fault tolerance, including checkpointing application state, restarting failed processes, and allocating additional nodes when needed. Finally, NR-MPI is a non-stop and fault resilient MPI built on top of MPICH that implements the semantics of FT-MPI [9]. These proposals hide the complexities of repairing the MPI state, however, they still rely on the programmers to instrument and modify the application code to obtain fault-tolerance support, including the responsibility of identifying which application data should be saved and in which points of the program. In contrast, the CPPC resilience proposal provides a transparent solution in which the application code is automatically instrumented by the CPPC compiler adding full fault tolerance support, both for detecting failures and repairing the MPI inner state as well as for checkpointing and recovering the application data. The fact that this proposal provides a transparent solution is specially useful for those scientific applications already developed over the years in HPC centers, in which manually adding fault tolerance support by programmers is, in general, a complex and time-consuming task.

3 CPPC overview

CPPC [19] is an application-level open-source checkpointing tool for MPI applications available under GPL license at `http://cppc.des.udc.es`. It 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. This instrumentation allows the application to periodically save the computation state into checkpoint files that can be used for its recovery after a failure.

CPPC implements several optimizations to reduce the checkpointing overhead [7]. The checkpoint file sizes are reduced by using a liveness analysis to save only those user variables indispensable for the application recovery; and by using the zero-blocks exclusion technique, which avoids the storage of memory blocks that contain only zeros. Besides, a multithreaded dumping overlaps the checkpoint file dumping to disk with the computation of the application.

Also, another CPPC feature is the portability. Applications can be restarted on machines with different architectures and/or operating systems than those in which the checkpoint files were generated. Checkpoint files are portable because of the use of a portable storage format (HDF5 `http://www.hdfgroup.org/HDF5/`) and the exclusion of architecture-dependent state from checkpoint files. Such non-portable state is recovered through the re-execution of the code responsible for its creation in the original execution. This is specially useful in heterogeneous clusters, where this feature enables the completion of the applications even when those resources that were being used are no longer available or the waiting time to access them is prohibitive.

3.1 Stop-and-restart proposal

The original proposal of CPPC to provide fault tolerance to MPI applications consist in a stop-and-restart checkpointing strategy [19]: during its execution the application periodically saves its computation state into checkpoint files, so that, in case of failure, the application can be relaunched and its state recovered using those files. As commented before, the CPPC compiler automatically instruments the application code to obtain an equivalent fault-tolerant version by adding calls to the CPPC library. The resulting fault tolerant code for the stop-and-restart proposal can be seen in Fig. 1. Instrumentation is added to perform the following actions:

- **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 the variables 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.

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

  MPI_Comm_split(MPI_COMM_WORLD, …, NEW_COMM);
  <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 …);
  }
  […]
  <CPPC_Unregister() block>
  CPPC_Shutdown();
}
```

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

- **Checkpoint:** the `CPPC_Do_checkpoint()` routine dumps the checkpoint file. 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.

Upon a failure, the application is relaunched and the restart process takes place. Firstly, 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 parts: reading the checkpoint data into memory and reconstructing the application state. The reading is encapsulated inside the routine `CPPC_Init_state()`. The reconstruction of the 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. When the execution flow reaches the `CPPC_Do_checkpoint()` call where the checkpoint file was generated, the recovery process ends and the execution resumes normally. The compiler inserts control flow code (labels and conditional jumps using the `CPPC_Jump_next()` routine) to ensure an ordered re-execution.

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. 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.

3.2 Resilience proposal

The use of traditional stop and restart fault tolerance solutions in high performance computing clusters presents several disadvantages. First, in these solutions the application is aborted in the event of a failure and a new MPI job needs to be relaunched. In some systems this implies the re-queueing of a new job to the scheduling system, introducing an overhead dependent of the cluster availability of resources. In the general case, the re-queueing will result in the assignment of a different set of resources, forcing the movement of all the checkpoint data across the cluster in order to restart the computation, usually causing significant network contention and, therefore, high overheads. However, a complete restart is unnecessary, since most of the computation nodes used by a job will still be alive and thus, more efficient solutions need to be explored. CPPC has been extended using the new functionalities provided by ULFM to transparently obtain resilient MPI applications [14], that is, applications that are able to recover themselves from failures without stopping their execution.

The CPPC resilience proposal maximizes its applicability by implementing a non-shrinking approach (preserving the number of running processes after a failure) and by recovering the application state using a global and backward strategy (in which the state of all the MPI processes is restored using the most recent valid recovery line). This solution is obtained by adding new instrumentation blocks in the application code, as shown in Fig. 2, to perform the following actions:

- Failure detection: the default error handler on each communicator is set to `MPI_ERRORS_RETURN` and 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. Once a process detects a failure, it revokes all of its communicators to ensure global failure knowledge, i.e., all the surviving processes will detect the failure.
- Reconfiguration of the global communicator: once all the surviving processes detect the failure, within the `CPPC_Check_errors()` routine, they invoke the `MPI_Comm_shrink()` routine, agreeing about the subset of failed processes and shrinking the global communicator, i.e. excluding the failed processes. Then, failed processes are re-spawned and the global communicator is reconstructed so that each surviving process will keep its original rank, while each one of the re-spawned ones will take over a failed process. During the restart, the application will obtain this new global communicator by means of the `CPPC_Get_comm()` routine.
- Restart of the application: a regular restart takes place, that is, the most recent valid recovery line is identified, checkpoint files are read, and an ordered re-execution of RECs reconstructs the application state. Thus, all processes must go back to the beginning of the application code so that they can re-execute the necessary RECs. This is done by performing the reversed conditional jumps introduced by the `CPPC_Check_errors()` and the `CPPC_Go_init()` instrumentation blocks.

```
int error; MPI_Comm GLOBAL_COMM;

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)
  <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;
  }
  […]
  <CPPC_Unregister() block>
  CPPC_Shutdown();
}
```

(a) CPPC instrumentation.

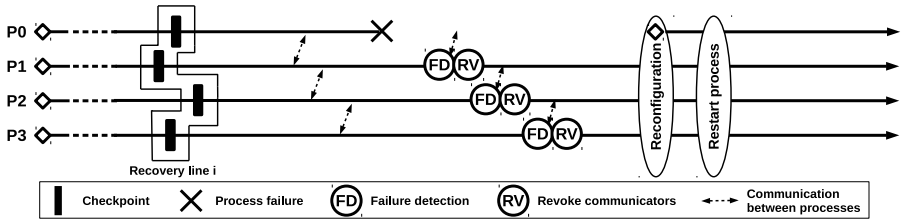
```
bool CPPC_Check_errors(int error_code){
 if( error_code == failure ){
   //Revoke communicators
   For c in application_communicators
     MPI_Comm_revoke(…, c, …)

   //Shrink global communicator
   MPI_Comm_shrink(…)
   //Re-spawn failed processes
   MPI_Comm_spawn_multiple(…)
   //Reconstruct communicator
   MPI_Intercomm_merge(…)
   MPI_Comm_group(…)
   MPI_Group_incl(…)
   MPI_Comm_create(…)

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

bool CPPC_Go_init(){
   return recovering_from_failure;
}
```

Failure detection

Reconfiguration of the global communicator

Start the recovery

(b) CPPC new routines.



(c) Behaviour of the resilient application.

Fig. 2: Resilient MPI applications combining CPPC and ULFM.

For more details about the resilience proposal combining CPPC and ULFM the reader is referred to [14].

4 Experimental evaluation

The experimental evaluation was performed at CESGA (Galicia Supercomputing Center) in the FinisTerrae-II supercomputer, comprised of nodes with two Intel Xeon E5-2680 v3 @ 2.50GHz processors, with 12 cores per processor and 128 GB of RAM, interconnected to an InfiniBand FDR 56Gb/s. The experiments were run spawning 24 MPI process per node (one per core). The CPPC version used was 0.8.1, working along with HDF5 v1.8.11 and GCC v4.4.7. The OpenMPI version used was ULFM commit a1e241f816d7. Finally, the Portable Hardware Locality (hwloc) [6] is used for the binding of the processes to the cores. Applications were compiled with optimization level O3.

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 and it was run setting the parameter `NRUNS` to 24×2^{16}. The Himeno benchmark [10] is a Poisson equation solver, it was run fixing NN to 24000 and using 2048x2048x1024 as grid size.

	Original runtimes (minutes) # of MPI processes				Total ckpt file size (GB) # of MPI processes			
	384	**768**	**1536**	**3072**	**384**	**768**	**1536**	**3072**
Sphot	60.3	30.4	15.6	8.9	0.3	0.5	1.0	2.0
Himeno	77.4	39.1	19.5	10.7	165.1	166.2	168.6	170.6
Mocfe	155.9	64.5	29.7	10.6	160.2	146.4	153.2	136.0

Table 1: Original runtimes and checkpoint file sizes for the testbed benchmarks.

Finally, MOCFE-Bone [23] simulates the main procedures in a 3D method of characteristics (MOC) code. It was run using 4 energy groups, 8 angles, a mesh of 28^3 doing strong scaling in space, and a trajectory spacing of $0.01cm^2$.

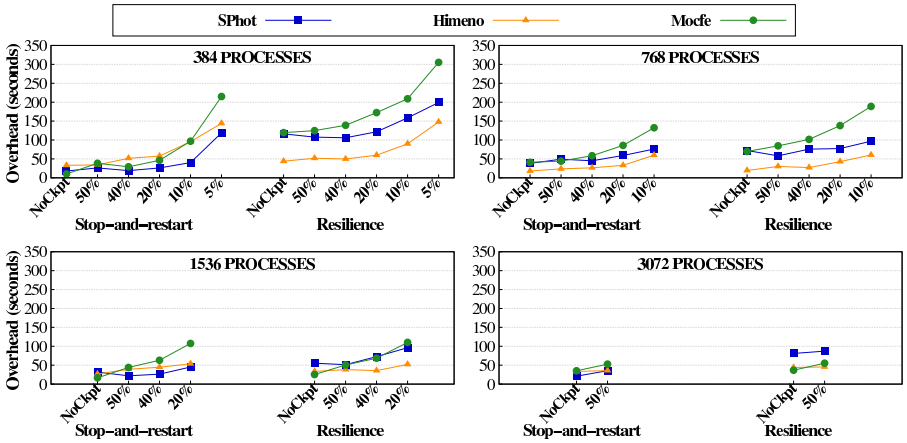
Table 1 shows, for each application and varying the number of MPI processes, the original runtime (without fault tolerance support), and the total checkpoint file size generated when one checkpoint is taken, that is, the addition of the individual checkpoint file size generated by each process. The reminder of this section evaluates and compares the stop-and-restart and the resilience versions of the applications. For a fair comparison, the same Open-MPI version was used in all tests. In both proposals, checkpoint files are stored in a remote disk using the Lustre parallel file system over InfiniBand.

4.1 Operation overhead in the absence of failures

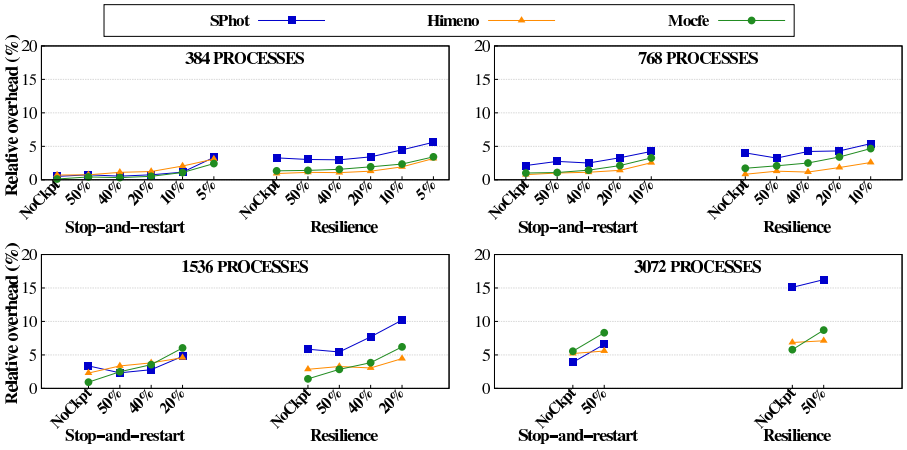
In a failure-free scenario, two main sources of overhead can be distinguished when using CPPC: the instrumentation and the checkpointing overheads.

The instrumentation overhead corresponds to the CPPC instrumented applications without generating any checkpoint files. Fig. 3a presents the absolute instrumentation overhead (in seconds), while Fig. 3b shows its relative value normalized with respect to the original runtimes, tagged as "NoCkpt" in both cases. As observed, the instrumentation overhead is larger when using the resilience proposal, which relies on a more extensive instrumentation, adding blocks of code around every MPI call for failure detection and backwards conditional jumping during the recovery. Differences between the instrumentations can be reduced using a MPI custom handler and non-local jumps [12], although workarounds are needed for its usage in Fortran applications.

The checkpoint overhead is measured in the execution of the CPPC instrumented versions generating checkpoint files, and it includes both the instrumentation overhead and the time spent in all the operations done when checkpointing. Note that the multithreaded dumping implemented by CPPC is used both in the stop-and-restart and the resilience proposal, thus, the checkpointed data is dumped to disk in background, hidding most part of the checkpointing overhead. Fig. 3a shows the absolute checkpointing overheads (in sedonds) using different checkpointing frequencies, while Fig. 3b presents the equivalent relative values (normalized with respect to the original runtimes). The checkpointing frequency is a user-defined parameter in



(a) Absolute checkpointing overhead (seconds).



(b) Relative checkpointing overhead (normalized with repect to the original runtimes).

Fig. 3: Checkpointing overhead varying the checkpointing frequency.

CPPC. Table 2 shows the different testbed checkpointing frequencies used (e.g. 20% means checkpointing every time the 20% of the computation has been completed), specifying the number of checkpoint files generated and the time elapsed between two consecutives checkpoints in each case. Note that the checkpointing frequency is increased until checkpoints are generated every 3-5 minutes with each number of processes. The checkpointing operation presents no differences whether using the stop-and-restart or the resilience proposals. However, the checkpointing overhead is larger for the resilience proposal. This is explained because the checkpointing overhead also includes the instrumentation cost, which, as commented previously, is larger in the resilience proposal. As observed, when increasing the checkpointing frequency, more checkpoint calls are taken, and thus, the checkpointing overhead increases. All in all, the

	Elapsed time (minutes) between checkpoints for different checkpointing frequencies														
	50%			40%			20%			10%			5%		
	1 ckpt taken			2 ckpts taken			4 ckpts taken			8 ckpts taken			16 ckpts taken		
	SPHOT	HIMENO	MOCFE	SPHOT	HIMENO	MOCFE	SPHOT	HIMENO	MOCFE	SPHOT	HIMENO	MOCFE	SPHOT	HIMENO	MOCFE
384 procs.	31	39	80	25	32	64	13	16	33	7	9	17	4	5	9
768 procs.	16	20	33	12	16	26	6	8	14	3	4	7	–	–	–
1536 procs.	8	10	15	6	8	12	3	4	6	–	–	–	–	–	–
3072 procs.	5	5	5	–	–	–	–	–	–	–	–	–	–	–	–

Table 2: Testbed checkpointing frequencies and total checkpoint file size.

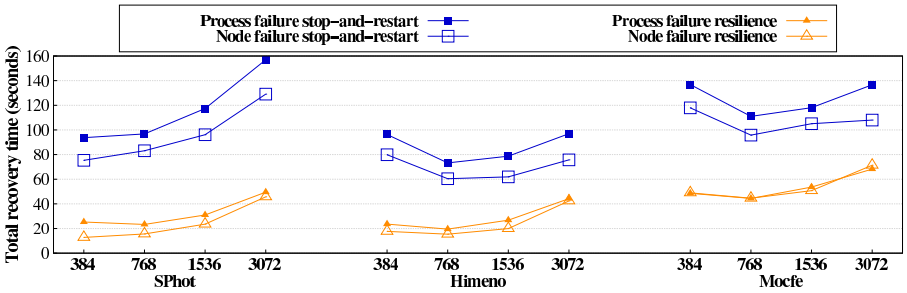


Fig. 4: Recovery time (seconds).

absolute overhead does not increase with the number of cores, while the relative overhead, which in general is below 5%, increases when scaling out the applications as the original runtimes decrease.

4.2 Operation overhead in the presence of failures

The performance of both the stop-and-restart and the resilience proposal is evaluated inserting one-process or full-node failures by killing the last ranked one or twenty-four 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. Table 3 summarizes the recovery operations performed in each proposal and described in Section 3. Fig. 4 presents for each proposal the addition of all the operations performed in each case to allow the application to continue its execution. On average, the resilience proposal reduces in 65% the recovery time of the stop-and-restart solution.

Fig. 5 breaks down the recovery operation times for each application. Failure detection times measure the time spent from the introduction of the failure until its detection. In the resilience proposal, it includes the time spent revok-

		Stop-and-restart	Resilience
Detection		Until application aborted due to failure	Until global knowledge of the failure (includes comm. revoke)
Other resilience operations (A)		–	Agreement about failed proc. (MPI_Comm_shrink)
Re-spawning		Application is relaunched, all processes re-spawned	Failed processes are re-spawned & initialized
Other resilience operations (B)		–	Global comm. reconstruction & backwards conditional jumps
Restart	**Reading**	Find recovery line and read checkpoint files	
	Positioning	Recover application state and positioning in the code.	

Table 3: Recovery operations in each proposal.

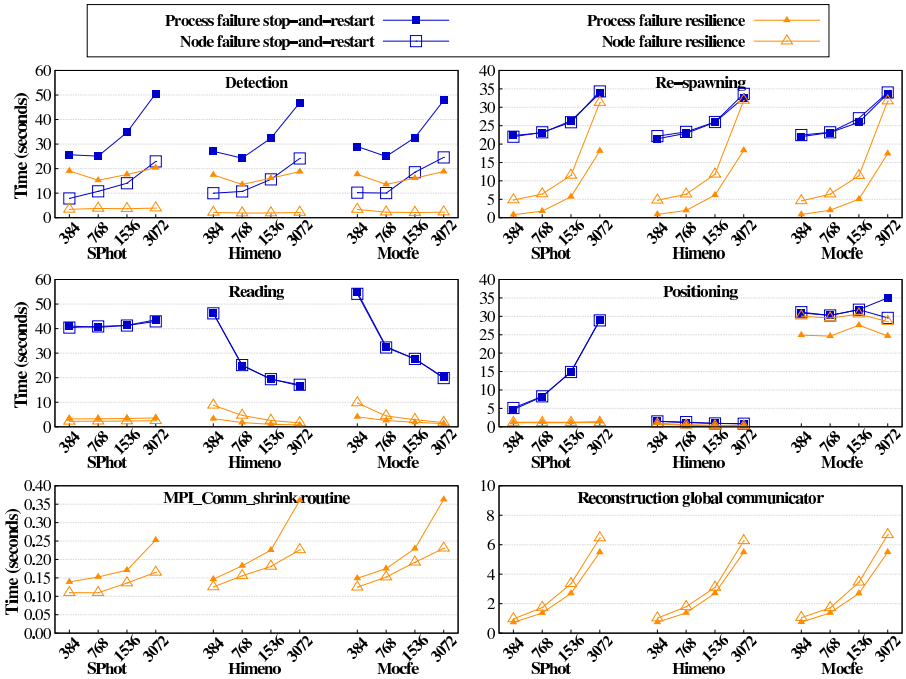


Fig. 5: Recovery operations times (seconds).

ing all the communicators in the application, which is inferior to 5 milliseconds in all the experiments. Detection times are better in the resilience proposal because of the detection mechanisms provide by ULFM. On average, detection is twice faster for one process failures and 6 times faster in the presence of node failures than when using the traditional stop-and-restart solution. In both proposals, as the number of failed processes increases, the time to detect the failure decreases.

In both proposals, the re-spawning times also include the initialization of the failed processes (time spent in the `MPI_Init` routine). The backwards

conditional jumps, with a maximum value of 0.5 milliseconds in all the tests, were not included in the figures. Note that, relaunching the entire application is always more costly than relaunching only the processes that have actually failed. However, this difference decreases as the number of failed processes increases, and, more important, when scaling out the application. As a result, the cost of re-spawning 24 processes when there are 3048 surviving processes, is close to the cost of relaunching 3072 processes from scratch.

The reading of the checkpoint files is faster in the resilience proposal, because the surviving processes benefit from the use of the page cache in which the checkpoint files from the most recent recovery line will frequently be present. In other scenarios, reading times can be reduced by using optimizations techniques, such as diskless checkpointing [22,17] in which copies of the checkpoint files are stored in the memory of neighbour nodes, or multilevel checkpointing [15,14], which saves those copies in different levels of the memory hierarchy. The restart positioning times are tight to the particular applications and the re-execution of the non-portable state recovery blocks. In SPhot positioning times are lower in the resilience proposal because the re-execution of these blocks benefits from the usage of page cache. Finally, the shrinking and the global communicator reconstruction are also represented in the figures. In both cases, these times increases with the number of processes running the applications. Shrinking times are larger when more survivors participate in the operation, as more survivors must agree about the subset of failed processes.

Note that, in both proposals the restart overhead would also include the re-execution of the computation done from the point in which checkpoint files were generated until the failure occurrence, an overhead that will be tight to the selected checkpointing frequency (more frequent checkpoints imply less re-execution overhead in the event of a failure, although more overhead is introduced during the fault free execution). Additionally, in some systems, the stop-and-restart proposal would also imply the re-queueing of a new job to the scheduling system, introducing an overhead dependent of the availability of the cluster resources.

5 Concluding remarks

This paper aims to assess the performance and compare two application-level fault-tolerant solutions for MPI programs: a traditional stop-and-restart approach and its equivalent resilience proposal using ULFM capabilities, with the focus on the scalability in current petascale systems.

The resilience solution clearly outperforms the stop-and-restart approach, reducing the time consumed in the recovery operations between 1.6x and 4x, and avoiding the resubmission of the job. During the recovery, the most costly steps are the failure detection and the re-spawning of failed processes. In the resilience proposal, the failure detection times are between 2x and 6x faster and the re-spawning times are also notably smaller. However, the re-spawning

times significantly increase when the number of failed processes grow and when scaling out the application. Thus, optimizations to minimize the re-spawning cost should be studied, such as the use of spare processes, initialized at the beginning of the execution that can take over the failed ranks upon failure [22].

The evaluation performed in this work is done on the basis of a general solution that can be applied to any SPMD code. However, ULFM allows for the implementation of different fault-tolerant strategies, depending on the nature of the applications at hand. Ad-hoc solutions could reduce the failure-free or the recovery overhead upon a failure. For instance, simplifying the detection of failures by checking the status of the execution in specific points, or avoiding the re-spawning of the failed processes in those applications that tolerate the shrinking of MPI processes.

Finally, in the evaluated resilience solution, all the application processes roll back to the last valid recovery line, thus, all processes re-execute the computation done from the checkpoint until the point where the failure have occurred. We believe that a global recovery should be avoided to improve the application performance both in time and energy consumption. Thus, as future work, we will explore this direction further considering the development of a message-logging protocol to avoid the roll back of the surviving processes.

Acknowledgements This work has been supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (project TIN2013-42148-P and pre-doctoral grant of Nuria Losada ref. BES-2014-068066) and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Units, co-funded by FEDER funds (Ref. GRC2013/055). We gratefully thank CESGA for providing access to the FinisTerrae-II supercomputer.

References

1. Ali, M. M. and Strazdins, P. E. and Harding, B. and Hegland, M.: Complex scientific applications made fault-tolerant with the sparse grid combination technique. International Journal of High Performance Computing Applications (2016)
2. ASC Sequoia Benchmark Codes: `https://asc.llnl.gov/sequoia/benchmarks/`. Last accessed: June 2016
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, p. 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, vol. 7490, pp. 193–203 (2012)
5. Bland, W., Raffenetti, K., Balaji, P.: Simplifying the Recovery Model of User-Level Failure Mitigation. In: Workshop on Exascale MPI at Supercomputing Conference, pp. 20–25 (2014)
6. 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: Int. Conference on Parallel, Distributed and Network-Based Computing (2010)
7. 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)

8. Di Martino, C., Kramer, W., Kalbarczyk, Z., Iyer, R.: Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5,000,000 HPC Application Runs. In: Int. Conference on Dependable Systems and Networks, pp. 25–36 (2015)
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/en/supercom/himenobmt/`. Last accessed: June 2016
11. Hursey, J. and Graham, R.L. and Bronevetsky, G. and Buntinas, D. and Pritchard, H. and Solt, D.G.: Run-through stabilization: An MPI proposal for process fault tolerance. In: Recent Advances in the Message Passing Interface, pp. 329–332 (2011)
12. 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)
13. Laguna, I. and Richards, D. F. and Gamblin, T. and Schulz, M. and de Supinski, B. R. and Mohror, K. and Pritchard, H.: Evaluating and extending user-level fault tolerance in MPI applications. Int. Journal of High Performance Computing Applications (2016)
14. Losada, N. and Cores, I. and Martín, M. J. and González, P.: Resilient MPI applications using an application-level checkpointing framework and ULFM. The Journal of Supercomputing pp. 1–14 (2016)
15. Moody, A. and Bronevetsky, G. and Mohror, K. and De Supinski, B. R: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Int. Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010)
16. Pauli, S. and Kohler, M. and Arbenz, P. : A fault tolerant implementation of Multi-Level Monte Carlo methods. In: Advances in Parallel Computing, pp. 471–480 (2013)
17. Plank, J.S., Li, K., Puening, M.A.: Diskless checkpointing. Transactions on Parallel and Distributed Systems, **9**(10), 972–986 (1998)
18. Rizzi, F., Morris, K., Sargsyan, K., Mycek, P., Safta, C., Debusschere, B., LeMaitre, O., Knio, O.: ULFM-MPI Implementation of a Resilient Task-Based Partial Differential Equations Preconditioner. In: Workshop on Fault-Tolerance for HPC at Extreme Scale, pp. 19–26 (2016)
19. 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)
20. 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: Int. Parallel and Distributed Processing Symposium, pp. 1225–1234 (2014)
21. Suo, G., Lu, Y., Liao, X., Xie, M., Cao, H.: NR-MPI: A Non-stop and Fault Resilient MPI. In: Int. Conference on Parallel and Distributed Systems, pp. 190–199 (2013)
22. Teranishi, K., Heroux, M.: Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In: European MPI Users' Group Meeting, pp. 51–56 (2014)
23. Wolters, E., Smith, M.: MOCFE-Bone: the 3D MOC mini-application for exascale research. Tech. rep., Argonne National Laboratory (2013)