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

# In-memory application-level checkpoint-based migration for MPI programs

**Iván Cores · Gabriel Rodríguez · María J. Martín · Patricia González**

Received: date / Accepted: date

**Abstract** Process migration provides many benefits for parallel environments including dynamic load balancing, data access locality or fault tolerance. This paper describes an in-memory application-level checkpoint-based migration solution for MPI codes that uses the Hierarchical Data Format 5 (HDF5) to write the checkpoint files. The main features of the proposed solution are: transparency for the user, achieved through the use of CPPC (ComPiler for Portable Checkpointing); portability, as the application-level approach makes the solution adequate for any MPI implementation and Operating System (OS), and the use of the HDF5 file format enables the restart on different architectures; high performance, by saving the checkpoint files in memory instead of disk through the use of the HDF5 in-memory files. Experimental results prove that the in-memory approach reduces significantly the I/O cost of the migration process.

**Keywords** Checkpoint · Migration · MPI · HDF5

## 1 Introduction

Process migration is the act of transferring a process from one system to another during its execution. It is an attractive feature as it enables: dynamic load balancing, by migrating processes from loaded nodes to less loaded ones; data access locality, by moving processes closer to the data that they are processing; and/or fault tolerance, by preemptively migrating processes from nodes that are about to fail.

Migration of processes in an MPI application is a greater challenge than that of a sequential version as special considerations regarding messages be-

Iván Cores · Gabriel Rodríguez · María J. Martín · Patricia González
Computer Architecture Group
University of A Coruña, Spain
E-mail: ivan.coresg@udc.es

tween processes have to be taken. In [1] the authors describe an application-level checkpoint-based approach to achieve process migration in MPI codes. The proposal is built on top of the CPPC framework [7], a portable and transparent checkpointing infrastructure for parallel applications. Using CPPC for the implementation introduces performance advantages. For instance, checkpoint file sizes are reduced thanks to the live variable analysis and zero-blocks exclusion performed by the CPPC compiler, which implies less memory requirements and a smaller checkpoint read/write overhead. Despite the reduced checkpoint file size, write/read of the process state to/from disk was determined as the main cause of overhead of the proposal. This work focuses on the reduction of this I/O cost.

The bandwidth of a hard disk is small when compared to network bandwidth, even when using RAID configurations. To take advantage of network speeds and avoid the bottleneck of disk accesses, a new migration approach is proposed in this work. The new solution substitutes storage in disk by in-memory checkpoint files and network transfers. Experimental results prove that the developed solution significantly reduces the overhead.

The structure of the paper is as follows. Section 2 presents the related work. Section 3 provides the appropriate background. In Section 4 the new migration approach is described. Section 5 evaluates the performance of the proposed solution, demonstrating its efficiency. Finally, Section 6 concludes the paper.

## 2 Related work

Process migration may be implemented either through dynamic migration or based on the simple stop-and-restart approach. In this section we will focus on proposals that, like the one seen in this work, address dynamic process migration.

Singh et al. [8] and Wang et al. [11] develop a system-level migration solution through checkpointing using the BLCR Linux Module. These proposals extend both BLCR and LAM/MPI to enable checkpoint and migration of only a few processes of the running set. A similar proposal that uses a different migration mechanism is MPI Mitten [3], a software solution that lays between the MPI implementation and the application. This proposal achieves independence from the underlying MPI implementation.

As regards the overhead costs of the migration mechanism, there exist in the literature several works that try to minimize the I/O overhead in process migration. In [6] a pipelined process migration with RDMA is presented. The proposed protocol pipelines checkpoint writing, and checkpoint transfer and read using data streaming through RDMA transport. Other recent solutions focus on the use of non-volatile memory technology, like solid-state disks (SSDs) [4], to store checkpoint data. SSDs offer excellent read/write throughput when compared to secondary storage and thus they can help reduce disk I/O load.

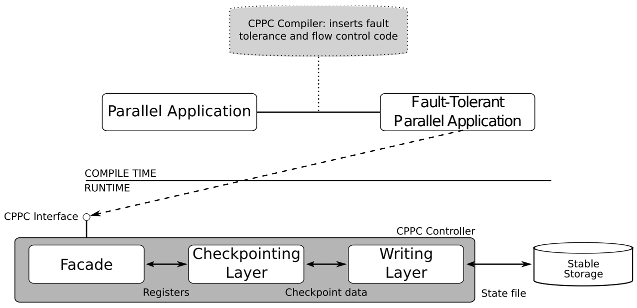


**Fig. 1** Integration of a parallel application with the CPPC framework.

Compared with the above proposals, the main advantage of our approach is that it is implemented at the application-level and thus it is independent of both lower-level layers, such as the OS or the MPI implementation used, and of any higher-level frameworks, such as job submission frameworks. Besides, application-level migration is able to obtain better performance by transferring only necessary data. The use of memory and network transfers instead of disk to store the checkpoint files reduces even more the overhead of our proposal. Finally, working at the application-level and using the 5th version of the Hierarchical Data Format (HDF5), a data format and associated library for the portable transfer of graphical and numerical data between computers, enable the restart on different architectures.

## 3 Background

### 3.1 CPPC overview

CPPC [7] is an application-level checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications. CPPC appears to the final user as a compiler tool and a runtime library. As shown in Figure 1, at compile time the CPPC source-to-source compiler automatically transforms a parallel code into an equivalent fault-tolerant version with calls to the CPPC library to instrument checkpointing. At runtime, the application sends petitions to the CPPC controller. From the structural point of view, the controller consists of three basic layers: a facade, that keeps track of the state to be stored when the next checkpoint is reached; the checkpointing layer, which gathers, manages and puts together all data to be stored into the state files; and a writing layer which decouples the other two layers from the specific file format used for state storage.

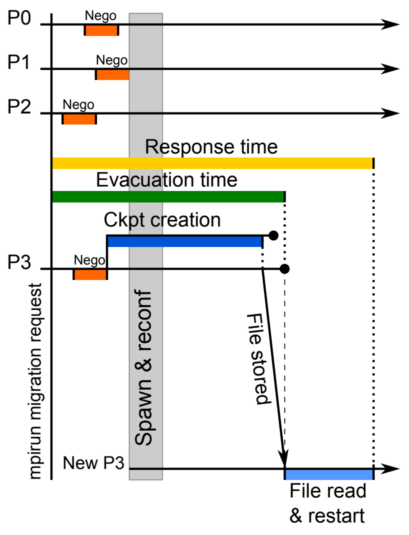
The CPPC framework solves several issues in implementing practical checkpoint solutions for parallel applications, such as checkpoint consistency, memory requirements and portability. As for the checkpoint consistency, the main handicap for parallel applications (compared with sequential ones) is the existence of dependencies imposed by inter-process communications. In order to avoid consistency problems caused by messages between processes, checkpoints are taken at locations where it is guaranteed that there are no in-transit, nor inconsistent messages. These locations are called safe points and are automatically detected by the CPPC compiler. Regarding memory requirements, CPPC reduces the amount of data to be saved by working at the variable level (i.e. storing user variables only) and performing a live variable analysis that identifies those variable values that are needed for the correct restart of the execution. Additionally, CPPC applies a size reduction technique called *zero-blocks exclusion* [2], which consists in avoiding the storage of memory blocks that contains only zeros. Working at the variable level allows to store only portable data, being possible to use portable file formats, like HDF5 [9], hence making restart possible on different architectures.

However, storing only portable data on state files has an important drawback: at restart time, not only user variables must be recovered, but also non-portable state created in the original execution. CPPC uses code reexecution to achieve complete application state recovery. A piece of code is defined as Required-Execution Code (REC) if it must be re-executed at restart time to ensure correct state recovery. The compiler will perform a source-to-source transformation, automatically identifying both the variables to be dumped to the checkpoint file and the non-portable code to be re-executed upon restart; and inserting the necessary CPPC functions, as well as flow control code. Thus, the restart in our proposal is divided into two steps: checkpoint file read, and effective state recovery. During the latter, the execution of certain RECs achieves the regeneration of non-portable state. For more details the reader is referred to [7].

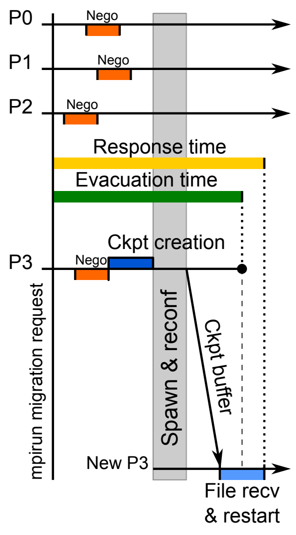
## 3.2 Disk-based migration using CPPC

The basic idea behind dynamic migration in MPI parallel applications is to spawn new processes that will be in charge of continuing the work of the terminating processes on other computing nodes. Communication groups must be rebuilt to exclude terminating processes and include the newly spawned ones.

The reconstruction of the communication groups is a critical step, since replacing communicators may lead to an inconsistent global state (messages sent/received using the old communicators cannot be received/sent using the new ones). The solution proposed in [1] is to perform the reconstruction of the communicators, and thus the migration, in locations where there are no pending communications, i.e. safe points. The CPPC compiler automatically detects safe points. Besides, based in a heuristic evaluation of computational cost, it places calls to the checkpoint function in selected safe locations. A negotiation protocol is built during runtime to select a single checkpoint location as the place to trigger the migration operation to achieve a consistent global state after migration. The algorithm designed is based in a forward negotiation, in which processes agree to coordinate at the next checkpoint call to be



(a) Disk solution



(b) Memory solution

**Fig. 2** Checkpoint-based migration with CPPC.

reached by the process that has advanced the farthest in the execution. This point will be referred to as *migration point*. One-sided MPI communications were used for the implementation, allowing processes to continue their regular execution during the negotiation.

Once the consensus on the migration point is achieved, the migration can start. In this phase, there are several sub-steps to be performed: saving the terminating processes state; spawning new processes to continue the execution in new nodes; updating the communicators between the processes and restoring the terminating processes state in the newly spawned processes. These steps are shown in Figure 2a) for four processes running on different nodes, one of which is having its process migrated to a new node. The dumping of the process state to disk is managed by a new ad-hoc thread using the CPPC capabilities, which allows for the reconfiguration to occur concurrently. The spawn is done by the `MPI_Comm_spawn_multiple` collective function. When spawning new processes, an intra-communicator between the original and the new processes is created. The dynamic communicator management facilities provided in MPI-2 are used to reconfigure the communicators and build a new global communicator (MPI_COMM_WORLD) replacing the intra-communicator. Other communicators built during the execution, which must derive from MPI_COMM_WORLD, will be reconstructed by re-executing the MPI calls used for creating them in the original execution (the CPPC restart capabilities described in Section 3.1 are used in this step). Terminating processes participate in the reconfiguration of the world communicator and wait until the creation of their checkpoint file is completed. When this happens, they notify the spawned processes that checkpoints may now be read

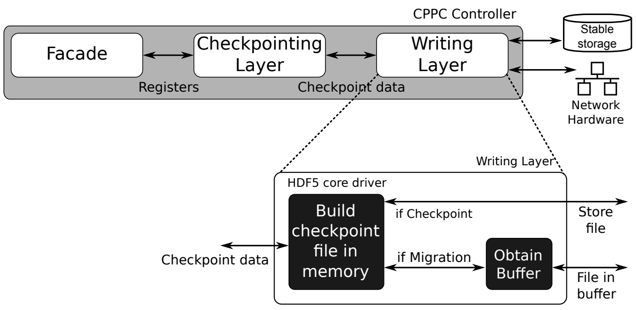


**Fig. 3** Modification of the CPPC Writing Layer

and finish their execution. The new processes wait until the checkpoint files are completely stored to disk. Then, they recover the saved state by reading the appropriate checkpoint file and executing the necessary RECs to regenerate non-portable state. This is achieved by delegating to CPPC and employing its native capabilities.

## 4 In-memory checkpoint-based migration

The most important disadvantage of the migration technique described in the previous section is the high time required to write and read the checkpoint files from disk. To reduce this overhead, in this new proposal the checkpoint files are stored in memory instead of in disk and sent to the new nodes using the network via MPI functions. In-memory HDF5 files are used for the implementation.

The steps followed by the in-memory checkpoint-based solution for four processes running on different nodes, one of which is having its process migrated to a new node, are shown in Figure 2b). The first step is the negotiation to reach a consensus on the migration point. The same protocol as the one presented in [1] is used for this phase. Then, the migrating processes save their process state, storing it in memory (Ckpt creation in the figure). In this case a new thread is not created, as writing checkpoint files in memory is very fast and I/O devices are not used. Afterwards, new processes are spawned in the target nodes to replace the migrating ones and the global communicator is reconstructed using the same MPI-2 functions as the ones mentioned in the previous section. Afterwards, the checkpoint files of the terminating processes are sent using MPI communications. In this point the terminating processes can safely finalize. The new processes receive the checkpoint files via an MPI communication and CPPC is used to recover the stored state.

For the implementation of this new approach, the CPPC writing layer (see Section 3.1) has been modified. The HDF5 library provides different file drivers which map the logical HDF5 address space to different types of storage. In the current CPPC version the default file driver (`SEC2 driver`) is used to dump the HDF5 data directly to stable storage. This driver is now substituted by the HDF5 `core driver` which allows to construct the HDF5 files in memory. Additionally, the `HDF5 File Image Operations` [10], available since HDF5 v1.8.9, are used to work with this files in the same ways that users currently work with HDF5 files on disk. The new writing layer is shown in Figure 3. When a signal with a migration request is received, the process state is stored in memory using the HDF5 core driver. Then, the HDF5 `H5Fget_file_image` function is used to obtain a buffered copy of these data in memory to be used as a file. This function returns a pointer to the buffer and its size. This information is used to send the data through the network to the newly spawned process using the MPI library. The data are received in a memory buffer in the newly spawned process via an MPI function. Finally, the `H5LTopen_file_image` function is used by the new process to convert the buffered memory to an HDF5 file. Then, CPPC is used to read the file and restore the application state using its standard mechanisms. The `HDF5 core driver` also allows to store the files directly to disk. This feature is used to store the files to disk when creating a checkpoint file in a non-migrating situation, thus enabling fault tolerance.

## 5 Experimental results

In this section the efficiency of the in-memory solution is evaluated. Comparisons with the disk-based approach are also shown. Two multicore clusters were used to carry out these experiments. The first one (Cluster N1 from now on) consists of eight nodes powered by two quad-core Intel Xeon E5620 CPUs with 16 GB of RAM. The cluster nodes are connected through a Gigabit Ethernet network. The working directory is mounted via network file system (NFS) and is connected to the cluster by the Gigabit Ethernet network. The second cluster (Cluster N2 from now on) consists of sixteen nodes powered by two octo-core Intel Xeon E5-2660 CPUs with 64 GB of RAM. These cluster nodes are connected through an InfiniBand FDR network. Finally, the working directory is mounted via NFS, and it is connected to the cluster by a Gigabit Ethernet network. The checkpoint files of the disk-based migration scheme were stored into the working directories. The files used in the in-memory scheme were sent directly between the nodes via the Gigabit Ethernet network in the first cluster and the InifiniBand FDR network in the second one.

The application testbed was composed of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [5] (NPBs from now on). The MPI implementation used was OpenMPI v1.5.4 for Cluster N1 and OpenMPI v1.6.4 for Cluster N2.

All the experiments were carried out using 16 and 32 processes and 8 processes per node (except BT and SP that need a square number of processes and were ran with 36) in order to also evaluate the scalability of the solution. The checkpoint file sizes for the NPBs using CPPC and 16 and 32/36 processes can be seen in Figures 4 and 5 (line labeled *Ckpt size* referenced to the right axis). The checkpoint sizes vary between 1.04 MB and 192.12 MB per

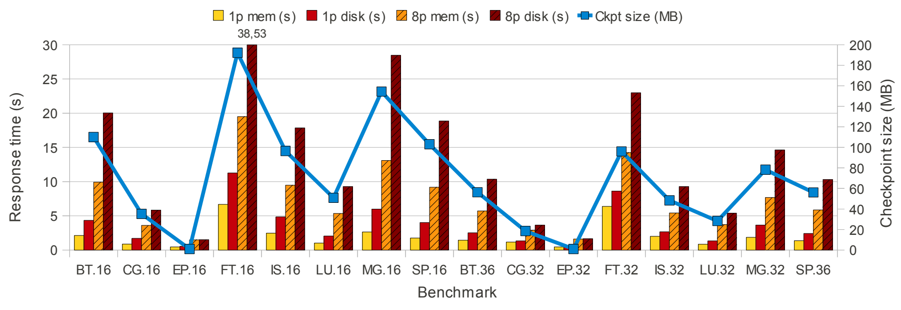


**Fig. 4** Response time (in seconds) in Cluster N1.

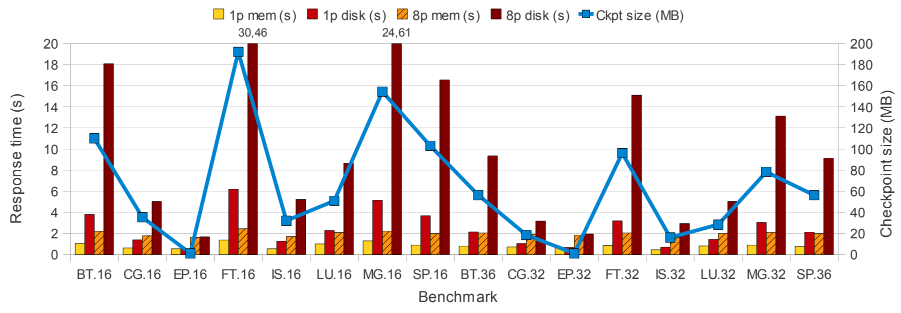


**Fig. 5** Response time for (in seconds) in Cluster N2.

process, which is an indicative that these benchmarks represent a wide range of applications.

## 5.1 Response Time

The response time is the execution time between the migration request and the actual restart in the new process (see Figure 2). Experiments in both clusters were carried out with two different configurations: migrating only one process and migrating the eight processes of a node. Figures 4 and 5 show response times in Cluster N1 and Cluster N2 respectively, migrating 1 and 8 processes via memory (*1p mem* and *8p mem* bars) and via hard disk (*1p disk* and *8p disk* bars).

The biggest contribution to the response time is the write and read of checkpoint files. Thus, the response time increases with the checkpoint file size and the number of migrating processes. When the total number of processes grows, the checkpoint files tend to become smaller and, thus, the response time decreases. For the smallest checkpoint files and migrating only 1 process the response time is negligible (see for instance EP).

Note that the response time via hard disk is always higher than the time needed to migrate via memory. The maximum benefits are obtained for applications with the large checkpoint files and, as expected, the differences are

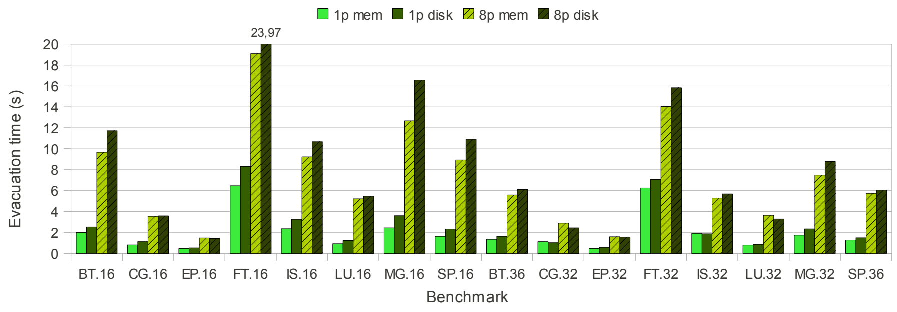


**Fig. 6** Evacuation time (in seconds) in Cluster N1.

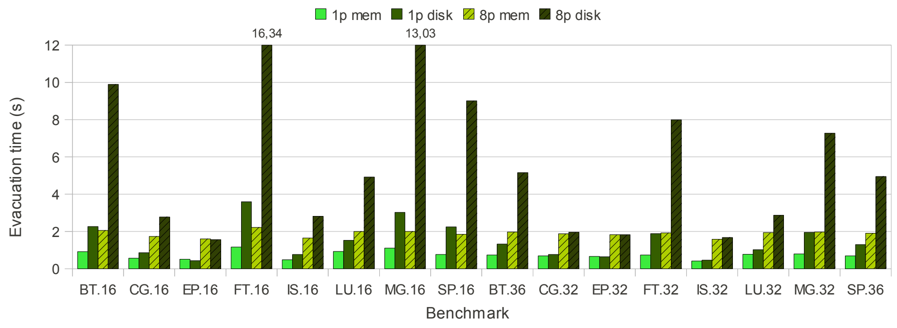


**Fig. 7** Evacuation time (in seconds) in Cluster N2.

more significant for Cluster N2 due to its faster network. For instance, the response time for FT with 16 processes, out of which 8 are migrated, decreases by 50% when using the in-memory solution over the Gigabit Ethernet network (Cluster N1) and by 90% over the InfiniBand one (Cluster N2).

## 5.2 Evacuation time

The evacuation time is the time needed after the migration request to safely finish the migrating processes (see Figure 2). Reducing this time is specially important when migration is used to implement proactive fault tolerance approaches (tasks are migrated in a preventive way when node failures are anticipated).

In the migration via hard disk the evacuation time is the time between the migration request and the dumping of the checkpoint files to stable storage. As for the in-memory migration, the evacuation time is the time between the migration request and the return of the MPI function used to send the in-memory checkpoint file.

Figures 6 and 7 show the evacuation times in Cluster N1 and Cluster N2 respectively. As can be seen, the in-memory solution is always better than the disk-based one except for the smallest checkpoint sizes, being the difference

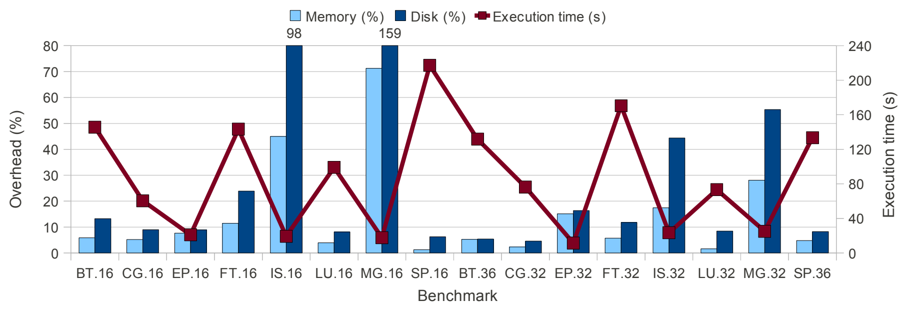


**Fig. 8** Overhead migrating 1 node (8 processes) in Cluster N1.

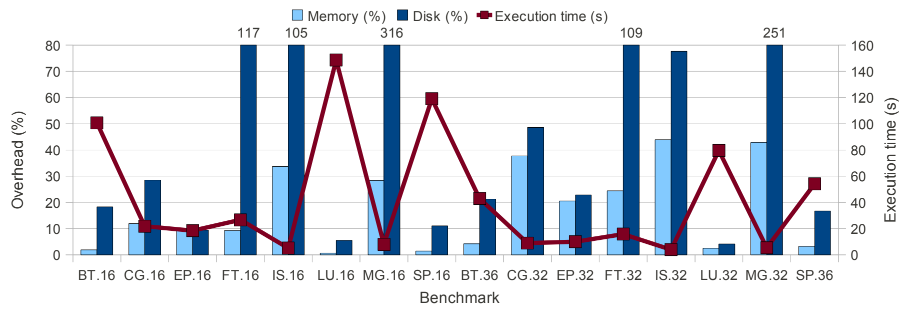


**Fig. 9** Overhead migrating 1 node (8 processes) in Cluster N2.

in these last cases not significant (less than 0.5 seconds). The best results are obtained for applications with large checkpoint files using Cluster N2. For instance, the evacuation time for FT in Cluster N2 with 16 processes, out of which 8 are migrated, decreases from 16.34 s for the disk-based solution to 2.22 s for the in-memory one (an 86%).

## 5.3 Overhead

The total execution times of the NPBs were measured to analyze the overhead introduced by the migration solutions. Figures 8 and 9 show the original execution times (line labeled *Execution time* referenced to the right axis) and the overheads in percentages of the memory-based (*Memory* bars) and disk-based (*Disk* bars) proposals for 16 and 32/36 processes in Cluster N1 and Cluster N2 respectively and migrating 8 processes. Note that the highest overheads are due to small execution times and large checkpoint files. As expected, the in-memory migration approach is always faster than the migration via hard disk. The benefit is more pronounced for fast networks and applications with large checkpoint files. For instance, for the FT application in the Cluster N2 with 16 processes the overhead decreases from 31.26 s to only 2.47 s.

## 6 Concluding remarks

Checkpoint-based migration uses checkpoint files and restart mechanisms to implement migration solutions. Checkpoints are typically stored to stable storage and in these cases checkpoint solutions are limited by the hard disk maximum I/O bandwidth. In this paper a proposal that transfers HDF5 checkpoint files directly to the remote memory without storing them to stable storage is presented. This in-memory approach moves the bottleneck of the system to the network bandwidth, reducing in all cases the overhead and the evacuation time. The obtained benefits are more significant for fast networks.

The in-memory migration has been implemented in CPPC and, therefore, it maintains the application-level features of the CPPC tool: independence of the OS, the MPI implementation used, and any higher-level framework.

**Acknowledgements** This research was supported by the Ministry of Science and Innovation of Spain (Project TIN2010-16735) and by the Galician Government (Project 10PXIB 105180PR).

## References

1. I. Cores, G. Rodríguez, P. González, and M. J. Martín. Failure avoidance in MPI applications using an application-level approach. *The Computer Journal*, 2012.
2. I. Cores, G. Rodríguez, P. González, and M. J. Martín. Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. In *Proceedings of ISPA 12*, pages 371–378, Madrid, Spain, 10–13 July 2012. IEEE Computer Society Press, Los Alamitos.
3. C. Du and X-H. Sun. MPI-Mitten: Enabling migration technology in MPI. In *Proceedings of CCGRID 06*, pages 11–18, Singapore, 16–19 May 2006. IEEE Computer Society Press, Los Alamitos.
4. M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. M. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA,* 13–19 November 2010, pages 1–12.
5. National Aeronautics and Space Administration. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. Last accessed July 2013.
6. X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. K. Panda. High performance pipelined process migration with RDMA. In *Proceedings of CCGRID 11*, pages 314–323, Newport Beach, CA, USA, 23–26 May 2011. IEEE Computer Society Press, Los Alamitos.
7. G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo. CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. *Concurr. Comput.-Pract. Exp.*, 22(6):749–766, 2010.
8. R. Singh and P. Graham. Performance driven partial checkpoint/migrate for LAM-MPI. In *Proceedings of HPCS 08*, pages 110–116, Québec City, Canada, 9–11 June 2008. IEEE Computer Society Press, Los Alamitos.
9. The HDF Group. HDF-5: Hierarchical Data Format. `http://www.hdfgroup.org/HDF5/`. Last accessed July 2013.
10. The HDF Group. HDF5 File Image Operations. `http://www.hdfgroup.org/HDF5/doc/Advanced/FileImageOperations/HDF5FileImageOperations.pdf`. Last accessed July 2013.
11. C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the* $21^{st}$ *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2008*, pages 1–12, 2008.