Reducing the overhead of an MPI application-level migration approach[1]

Iván Cores, Mónica Rodríguez, Patricia González, María J. Martín

Computer Architecture Group
University of A Coruña Spain
Email: {ivan.coresg, patricia.gonzalez, maria.martin.santamaria}@udc.es

Abstract

Process migration provides many benefits for parallel environments including dynamic load balance, data access locality, or fault tolerance. This work proposes a solution that reduces the memory and I/O overhead in an application-level checkpoint-based migration approach. The proposal splits the checkpoint files in order to overlap the writing of the state in the terminating processes with the read and restarting operation in the newly spawned processes. It has been tested using the MPI NAS Parallel Benchmarks, showing encouraging results, both in terms of memory consumption and I/O migration times.

Keywords: migration, checkpointing, MPI

1. Introduction

Process migration consists 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.

Nowadays, many parallel applications are written using MPI. However, MPI does not provide explicit support for migration. In a previous work [2], the authors describe a checkpoint-based approach to achieve process migration in MPI codes. The proposal is implemented at the application-level using the CPPC checkpointing framework [3]. When a signal with a migration request is received, the state of each process to be migrated is stored into memory and transferred to a newly spawned process, which reads this file and recovers the state of the terminating process.

[1]This work is an extended version of a paper presented at the 26th International Symposium on Computer Architecture and High Performance Computing [1]

This work proposes the split of the generated checkpoint files to overlap the different phases of a migration operation (state file writing in the terminating processes, data transfer through the network, and state file read and restart operations in the new processes), thus, reducing the migration time. Moreover, the split of the state files also allows for reduction in memory consumption, since it avoids storing the complete checkpoint file. Both improvements will be especially important for those applications with large checkpoint files.

The structure of the paper is as follows. Section 2 presents related work. Section 3 presents an overview of CPPC, the checkpointing framework used for the migration of MPI processes. Section 4 details the design and implementation of the proposed approach to reduce the migration overhead. Section 5 evaluates the performance of the solution, focusing on the reduction obtained both in terms of migration time and memory consumption. 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 [4, 5]. This section will focus on proposals that, similar to the one proposed in this work, address dynamic process migration.

Some existing approaches rely on operating system virtualization techniques. The use of OpenVZ is investigated in [6] to perform dynamic migration of parallel applications. Charm++ [7] and Adaptive MPI (AMPI) [8] are used in [9] to implement a transparent proactive fault tolerance approach through migration. In [10] the live migration mechanism in Xen [11] is exploited to implement a live migration solution. However, in an HPC context, solutions at the process level are more widely accepted than those based on virtualization [12], mainly due to the lower penalty in performance.

In [13] a process level solution through checkpointing using a system-level migration based on the BLCR (Berkeley Lab's Checkpoint/Restart Library) [14] Linux Module is presented. The proposal extends both BLCR and LAM/MPI to allow process migration. Although the paper does not present experimental results, it explicitly states that the checkpoint file writing has a high overhead. In [15] this overhead is reduced through a live migration solution. In this approach, execution proceeds while a process image is asynchronously transferred to a spare node. Its main disadvantage is the increase in the evacuation time of the terminating processes. The migration mechanism implemented in MVAPICH2 [16] also relies on BLCR. It takes advantage of the Remote Direct Memory Access (DRMA) over InfiniBand to reduce the I/O overhead [17]. A similar proposal that uses a different migration mechanism is MPI Mitten [18], an MPI library implemented on the HPCM (High Performance Computing Mobility) middleware [19] which achieves some independence from the underlying MPI implementation. All these solutions are based on a coordinated checkpointing approach to reach a consistent global state.

Checkpointing is, thus, the most popular solution to perform migration at the process level. However, its cost in terms of network utilization or memory consumption can be a limiting factor for large applications. Since the checkpoint file size is the most important factor in determining checkpointing performance, a widespread solution to minimize the checkpointing cost is to reduce the checkpoint file size. Techniques to reduce the checkpoint file size present in the literature, such as data compression of the checkpoint files [20, 21], or memory exclusion [22], could be used to improve migration solutions. Another way to optimize the I/O cost of migration based on checkpointing is to overlap the I/O operations. In [23] a pipelined process migration using data streaming through RDMA transport is presented.

The approach proposed in this paper also provides a process level checkpoint-based solution. Its main advantage, when compared with previous works, is that it is implemented at the application–level, thus, providing several benefits. It is independent of lower-level layers, such as the OS or the MPI implementation used, and of any higher-level frameworks, such as job submission frameworks. Besides, by storing only portable data in a portable manner, it enables the restart on different architectures.

Additionally, our proposal comprises different techniques to reduce the overhead. First, CPPC already implements different mechanisms for reducing the checkpoint files sizes [24]. Second, as a result of splitting the checkpoint files, the write and read of process state can be overlapped, thus minimizing migration time and memory overhead.

3. CPPC Framework

CPPC (ComPiler for Portable Checkpointing) [3] is an application-level checkpointing tool focused on the insertion of fault tolerance into long-running message-passing aplications. It is available under GNU general public license (GPL) at http://cppc.des.udc.es.

CPPC appears to the user as a compiler tool and a runtime library. As shown in Figure 1, at compile time the CPPC compiler automatically transforms a parallel code into an equivalent fault-tolerant version with calls to the CPPC library to instrument checkpointing.

The CPPC framework solves several issues in implementing practical checkpoint solutions for parallel applications, such as checkpoint consistency, memory requirements and portability.

As for checkpoint consistency, in order to avoid consistency problems caused by messages between processes, checkpoints are taken at locations where it is guaranteed that there are neither in-transit nor inconsistent messages. These locations are called safe points. The CPPC compiler automatically detects safe points. Besides, based in a heuristic evaluation of computational cost, it identifies the most expensive loops in the application code and inserts a checkpoint call (`CPPC_Do_checkpoint()`) in the first safe point of these loops [25]. However, not all checkpoint function calls will generate checkpoint files. In a traditional

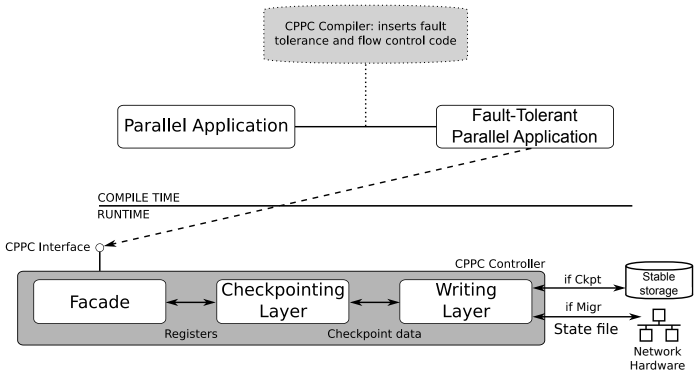

Figure 1: CPPC overview

fault tolerance context, a checkpoint file will be generated every N calls to this function, being N user-defined. In a migration context, the checkpoint dumping will be triggered when a signal with a migration request is received.

Regarding memory requirements, CPPC implements different mechanisms for reducing the checkpoint files sizes [24]. It reduces the amount of data to be saved by working at the variable level 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 which consists in avoiding the storage of memory blocks that contains only zeros. Also, CPPC allows the compression of the checkpoint files to remove redundant information. Besides, it provides multithreaded dumping, overlapping the checkpoint file writing with the computation of the application.

Working at variable level allows both to reduce the amount of data to be saved and to store only portable data, hence making restart possible on different architectures. The variables that need to be stored into state files are explicitly marked by the compiler at compilation time. This process is called variable *registration*.

However, storing only portable data in state files involves extra work at restart time, when non-portable state created in the original execution must also be recovered. 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 automatically identify both the variables to be dumped to the checkpoint file and the non-portable code to be re-executed upon restart; and insert the necessary CPPC functions and flow control code. Thus, the restart is divided into two steps: checkpoint file read and effective state recovery.

CPPC uses a configuration file to set certain parameters such as checkpoint frequency or compression type. A default file is provided, but it can be modified to specify a different behavior.

For more details about CPPC and its restart protocol, the reader is referred to [3, 26].

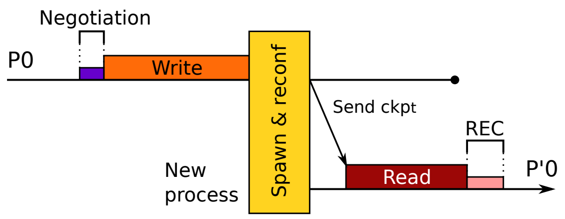

Figure 2: CPPC migration with sequential write-transfer-restart steps

3.1. Migration operation

Dynamic migration in parallel applications is based on spawning new processes that will be in charge of resuming the work of the terminating processes on other computation nodes. In a checkpoint-based solution, when a signal with a migration request is received, the terminating processes write their state to checkpoint files, while each newly spawned process read the respective file and recovers the state of its corresponding process that is being terminated.

Before resuming the execution, communication groups must be rebuilt to exclude terminating processes and include the newly spawned ones. This step is crucial, since messages sent/received using old communicators cannot be received/sent using the new ones. The solution based on CPPC performs the migration in locations where there are no pending communications, i.e. in safe points. As commented above, those points are detected automatically by the CPPC compiler, which inserts calls to the `CPPC_Do_checkpoint()` function in selected safe locations. These calls could be used as migration points. However, conducting the migration from different checkpoint calls in different processes may lead to inconsistencies. Therefore, a negotiation protocol is performed at runtime to select a single checkpoint call as the place to trigger the migration operation to achieve a consistent global state after migration. The negotiation algorithm is included inside the `CPPC_Do_checkpoint()` function and executed only when a migration request is received.

Once the migration point is reached, the migrating processes save their process state. Then, new processes are spawned in the target nodes to replace the migrating ones and the global communicator is reconstructed. The dynamic process management facilities of MPI-2 are used for these operations. Afterwards, the checkpoint files of the terminating processes are sent using MPI communications. At this point the terminating processes can safely finalize. Then, the new processes recover the saved state by reading the checkpoint files and executing the necessary RECs to regenerate non–portable state. This is achieved by delegating to CPPC and employing its native capabilities. The migration process is depicted in Figure 2.

The multithread dumping provided by CPPC does not get to hide the I/O cost in a migration context, since the checkpoint dumping, and its transfer and read, cannot be overlapped with further computation, neither in the terminating process, nor on the new spawned one. Thus, the time required to write/read the checkpoint files to/from disk was determined to be the main cause of overhead during the migration operation. To take advantage of network bandwidth and avoid the bottleneck of disk accesses, in-memory checkpoint files and network transfers were proposed in [2]. The experimental results of this work show that the in-memory approach significantly reduces the I/O cost of the migration process. However, the significance of this reduction depends heavily on the network used. Additionally, it has large memory requirements, as a copy of the whole checkpoint has to be stored into memory. Thus, a further improvement is proposed in this paper to reduce both the memory and the I/O overhead.

4. Reducing the overhead in the migration process

In the proposal described in the previous section the writing of the state file, the transfer of this file to the new process and the read and recovery of the saved state are executed in a sequential way. When the size of the checkpoint files is large, the time due to the serialized execution of these three phases may become unreasonable. To reduce this time, these three steps could be executed in a pipeline fashion. To that end, the checkpoint files are split into multiple smaller files so that the new process can start with the read step while the original process continues to write other checkpoint fragments. In this way, the time that the new process has to wait to begin the restart operation is shortened, thus reducing the whole time of the migration operation. Moreover, less memory storage is required, since only those fragments that are being written and transferred in each step need to be temporarily stored.

Next section describes the structure of the checkpoint files generated by CPPC and how these files are split to make the pipeline operation possible.

4.1. Splitting the checkpoint files

In CPPC, checkpoint files are stored using HDF5 [27], a data format and associated library for the portable transfer of graphical and numerical data between computers. HDF5 files consist of two primary types of objects: groups and datasets. An HDF5 group is a container structure which can hold datasets and other groups. An HDF5 dataset is a multidimensional array of data elements. Both types support metadata. Any HDF5 group or dataset may have an associated attribute list, which is a user-defined HDF5 structure that provides extra information about an HDF5 object.

CPPC checkpoint files contain not only the relevant data needed to continue with the execution, but also all the context information needed for the correct restart of the application. They are structured hierarchically using HDF5 as depicted in Figure 3. Each state file is divided in two different parts: a metadata section and an application data section. The metadata section consists of three main parts, a dataset called Header and two groups: FileMap and Context. The dataset Header contains information about the checkpoint file, such as the compression type. The FileMap group records all the files opened during the execution. The Context group keeps track of execution context changes. Each context object represents a call to a procedure, and contains the information required for recovering data in that procedure scope. Contexts can contain

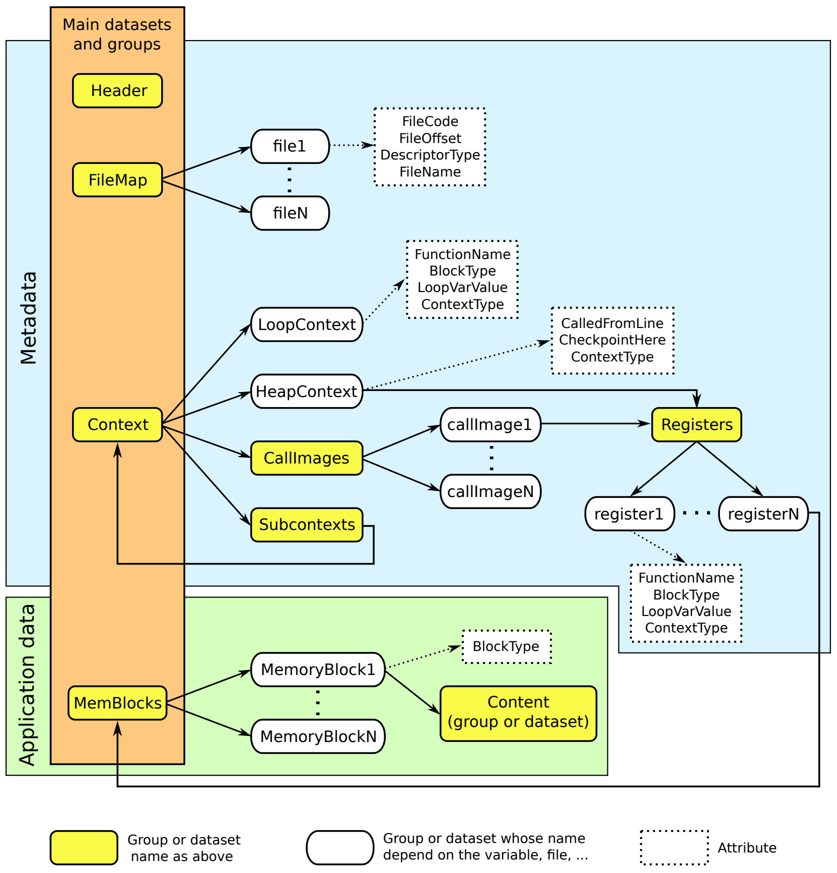

Figure 3: Structure of the CPPC checkpoint files

subcontexts, created by nested calls to the same or other procedures. This hierarchical representation allows for the sequence of procedure calls made by the original execution to be recreated upon restart. In this way, the application stack is rebuilt, and the relevant state is recovered inside its appropriate scope. Finally, the application data section contains the MemBlocks group, which includes the value of all the registered variables. This group contains a subgroup, MemoryBlock from now on, for each variable. Each subgroup includes one or more datasets. The MemBlocks group is the largest portion of the checkpoint file.

Considering the CPPC restart operation, the checkpoint files are split into two main parts. The first part includes three objects: Header, FileMap and Context. Once the new process receives this part, it can begin to rebuild the application stack and open the necessary files. Thus, this first part is transferred in a single fragment. Although the size of this fragment depends on the application, and, more specifically, on the context changes of the application, it is always negligible compared to the total checkpoint file size (inferior to 1% for all the applications used in this paper).

The second part contains the object MemBlocks, which represents the bulk of the checkpoint file. This part, in turn, can be split in multiple chunks. The maximum size of each chunk can be specified into the CPPC configuration file. Its default value is set to 64 MB, that demonstrated a good cost/efficiency relation in the experimental tests. Attending to the maximum size of each chunk, the MemBlocks group will be split at a different structure level:

- *MemoryBlock level*, when the MemBlocks group is split into two or more files, each one containing one or more MemoryBlocks.
- *Dataset level*, if a MemoryBlock contains several datasets, it can be split by grouping one or more datasets in different files.
- *Element level*, when each dataset is divided into two or more chunks. In this level, when a dataset represents a block of zeros, it will never be split, since, thanks to the zero-blocks exclusion technique applied by CPPC, it will only contain one element.

When the splitting is done at dataset or element level, some extra information has to be added to allow the new process to restore the MemoryBlock accurately. This information is inserted in the checkpoint file using three new defined HDF5 attributes:

- `isNotFirstChunk`, associated with de group Content, indicates that this MemoryBlock has already been created. Therefore, during restart, the new process should search for it rather than create a new one.
- `FragmentTotalElements`, included only in the first fragment of a fragmented dataset, it indicates total number of items in the dataset and it is used by the new process to allocate memory for the complete dataset.
- `PositionOfFragment`, indicates the starting position of the elements in the dataset. This attribute is included in all dataset chunks except for the first.

4.2. Implementation issues

To implement the overlap in writing, transfer and read phases of the migration process, the CPPC migration operation seen in Section 3 (see Figure 2) is modified as shown in Figure 4. Negotiation phase remains unchanged. However, the spawn of the new process and the reconfiguration of communicators is brought forward at the beginning of the operation. In this way, the original process is able to send the different chunks, step by step, instead of sending the complete state file at the end.

Additionally, the CPPC writing layer (see Figure 1) is modified to split the MemoryBlocks complying with the chunk size specified in the CPPC configuration file and to add the HDF5 attributes commented in previous subsection.

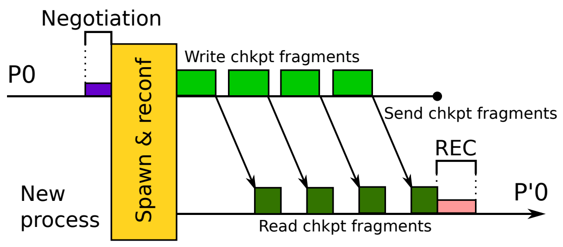

Figure 4: CPPC migration with pipelined write-transfer-read steps

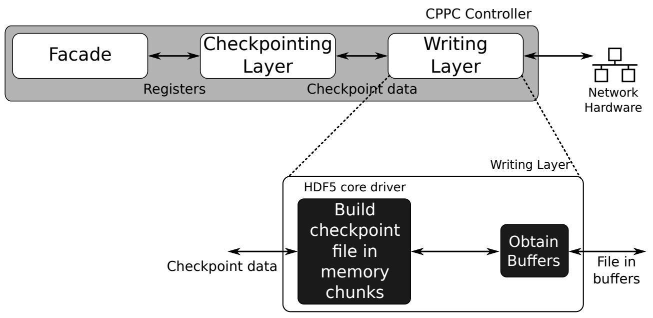

Figure 5: Modification of the CPPC writing layer

Now, every time a MemoryBlock is dumped, the writing function checks whether it should be split at dataset level, or even at element level. The new writing layer is depicted in Figure 5. The process state is stored chunk by chunk to memory using the HDF5 core driver. Then, a buffered copy of each chunk is sent through the network to the newly spawned processes using the MPI library. In each step, HDF5 starts to dump the next chunk at the same time that the chunk copied into the buffer is sent to the new process. Thus, the memory overhead is twice the chunk size.

The CPPC reading function is also modified. Now, whenever a chunk is read, it is necessary to check if the MemoryBlock has been split, and at which level. If it has been split, CPPC checks whether it is the first chunk or not. In the first case, space for the whole MemoryBlock will be reserved. In the second case, the position of that chunk in the MemoryBlock is stated.

After the checkpoint file read, the memory addresses for the registered variables will be back-calculated to point to the address containing its data. Therefore, the memory used to store the transferred chunks will be the final memory used by the restored variables, and there will be no memory overhead in the new spawned process.

Finally, the blocking MPI communications used for the transfer of the checkpoint files in the previous version [2] are replaced by non-blocking MPI calls so that they can be overlapped with writing and reading steps.

5. Experimental results

This section explores the efficiency of the proposed solution. A multicore cluster was used to carry out these experiments. It consists of nodes powered by two octo-core Intel Xeon E5-2660 with 64 GB of RAM. The cluster nodes are connected through an Infiniband FDR network. The MPI implementation used was OpenMPI v1.6.4.

The application testbed is composed of six out of the eight applications in the MPI version of the NAS Parallel Benchmarks [28] (NPBs from now on). The IS benchmark was discarded due to its low execution time and the EP benchmark due to the small size of its checkpoint files. The NPBs represent a set of typical and varied HPC application kernels, which makes them a good option for comparison purposes. For all the executions, the benchmark size used is class D.

All the experiments were carried out using 32 processes in 2 nodes (except BT and SP that need a square number of processes and, thus, they were executed with 36 processes using a third node). The experiments were focused on the performance evaluation both in terms of reduction in memory consumption and reduction in migration time with respect to the previous CPPC version [2]. The experiments were designed so that, for each application, the migration were triggered always at the same point for all the executions. We had exclusive access to the cluster during the run of the experiments.

5.1. Memory consumption

One of the most valuable goals associated to the proposed solution is that it requires considerably less memory space than the serialized classical version, which might be decisive to assure the feasibility of the migration.

Figure 6 shows the results of memory consumption per process for a 32-process execution of: an execution without migration (Mig. free); an execution performing one migration based on dumping the complete checkpoint file into memory before transferring it to the new spawned process (Original); and an execution performing one migration where the checkpoint file is split and the write, transfer and read phases are overlapped (Split). The increase in memory consumption during the migration is due to the memory requirements to save the state file. In the original non-split version, the increase in the memory consumption is according to the size of the checkpoint file per process (shown also in this figure). When the splitting technique proposed in this paper is applied, only the checkpoint data that corresponds with the segment that is being written or read at that moment need to be stored. In the experiments shown in this figure a chunk size of 64 MB (the default value) has been used. In this case, the actual memory requirements are double of the segment size, since, as commented in Section 4.2, while a copy of the written segment is being transferred, another segment begins to be dumped in the HDF5 buffer.

Note that the increase in memory consumption only arises in those processes that are being migrated. Thus, the memory consumption per computational node depends on the number of migrating processes in each node. Figure 7

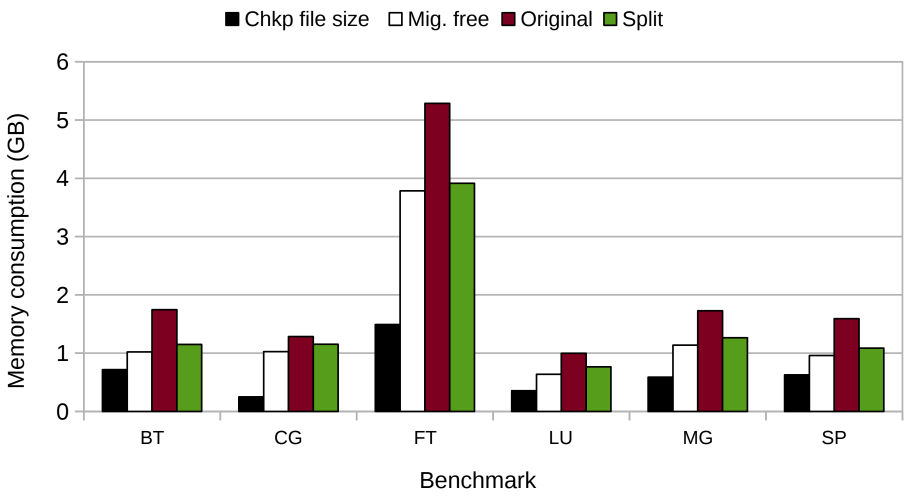

Figure 6: Checkpoint file size and memory consumption per process in a 32-process execution (in GB).

shows the memory consumption per node when different number of processes are migrated. The system used has 64 GB of RAM per node which means that FT (class D) application cannot be migrated using in-memory storage unless the splitting technique is employed. Therefore, the save in memory consumption is particularly relevant because, if the size of the complete checkpoint file is too large to fit in memory, the pipeline solution allows for avoiding the use of disk storage that would represent a significant bottleneck in the migration operation, as illustrated in next subsection.

5.2. Migration time

The migration time is the execution time between the migration request and the completion of the restart operation in the new process. For the original version, the migration time is broken down into 5 parts (see Figure 2):

- *Negotiation*: execution time between the mpirun migration request and the arrival to the migration point.
- *WriteCkpt*: time required for the checkpoint writing.
- *Spawn*: execution time of the spawn function and the reconfiguration of the communicators.
- *TransferRead*: time required for the transfer of the checkpoint files and the read of these files in the newly spawned processes.
- *Restart*: time required for the restart of the application once the checkpoint files have been read. It includes the execution of the RECs, as explained in Section 3.

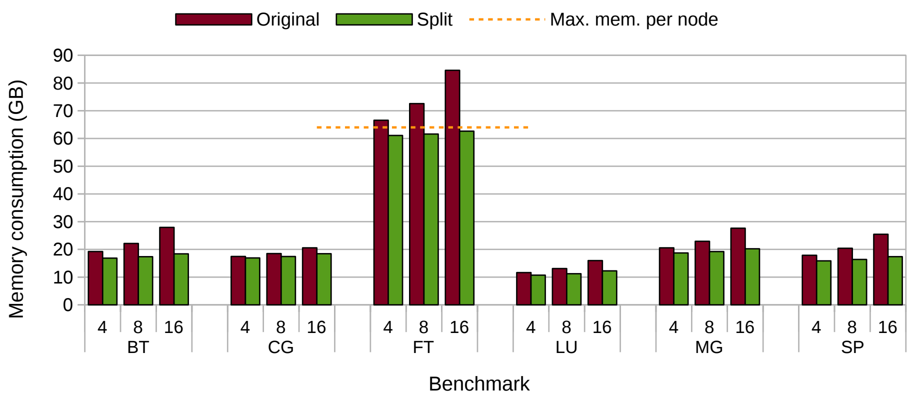

Figure 7: Memory consumption (in GB) per node (16 cores) in a 32-process execution when different number of the 16 processes per node are migrated.

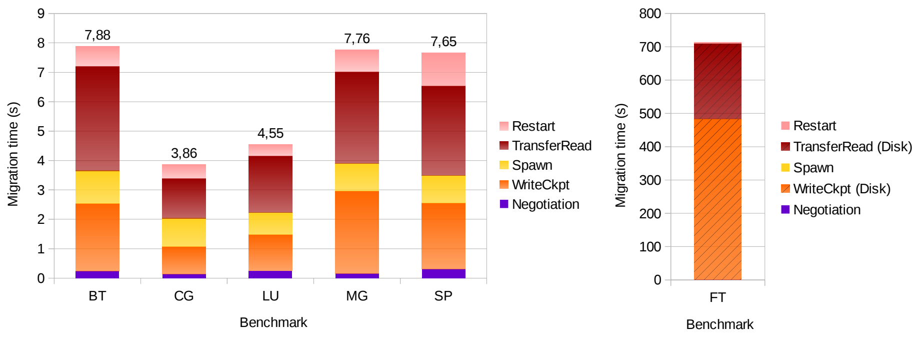

Figure 8: Migration time in the original CPPC version in a 32-process execution when 16 processes are migrated.

Figure 8 shows the migration time in the original CPPC approach in a 32-process execution when all the processes of a node (16 processes) are migrated. The breakdown of the migration time shown in the figure corresponds to the slowest migrating process.

The *Negotiation* time depends on: the frequency of the checkpoint function calls, as the negotiation algorithm is included inside this function; the inherent synchronization between the processes during the execution of the application, since the same checkpoint call has to be reached by all processes; and the overhead introduced by the negotiation protocol itself. To measure the worst case, the migration signal is received, by at least one of the processes, just after a checkpoint call. Thus, the negotiation time will be at least the time between two consecutive checkpoint calls. In all the NPB, the CPPC checkpoint function is called once in each iteration of the main computational loop. The processes of these applications are inherently synchronized in every internal iteration of the application. This means that, during the negotiation phase, one process will never advance more than one iteration before reaching the migration point. Results shown in Figure 8 prove that the overhead associated to the negotiation protocol is almost negligible, even in this worst case.

The *Spawn* time is almost constant for all the applications. The spawn function is a collective operation, thus, it depends on the total number of processes involved in the application execution.

The *Restart* time depends on the RECs that need to be executed, and, thus, it is dependent on the application. For all the benchmarks considered, the *Restart* time does not contribute significantly to the migration time.

The most impacting contributions to the migration overhead are: *WriteCkpt*, and *TransferRead*, despite the fact that checkpoint files are stored in memory. Note that the migration using the original approach in FT cannot be performed *in-memory* (see Table 7), and disk storage is used instead, thus, increasing significantly the *WriteCkpt* and *TransferRead* times.

In the new proposal *WriteCkpt* is overlapped with *TransferRead* in order to reduce the migration times. This time will be labeled as *WriteTransferRead* in the next figures and it corresponds to the time between the end of the reconfiguration phase in the newly spawned processes and the actual restart (see Figure 4). Note that the maximum improvement that can be achieved with the pipeline proposal is limited by the minimum between the *WriteCkpt* time and the *TransferRead* time in the original one.

5.2.1. Impact of the number of migrating processes

This subsection shows the improvement obtained in the write-transfer-read times with the split approach in function of the number of processes that migrate from one node. Note that the most realistic situation would be that in which several processes in one node have to be migrated (if the node is overloaded), or even the whole node needs to be migrated (if the node is about to fail). Figure 9 shows the results for each benchmark using 32 processes and migrating 4, 8 and 16 processes from the same node. The times shown in the figure correspond to the slowest migrating process.

The amount of data to be dumped into the node memory and to be transferred through the network increases with the number of migrating processes. Therefore, the overhead of the *WriteCkpt* and *TransferRead* phases in the original approach, and the *WriteTransferRead* step in the split solution increases when the number of migrating processes per nodes grows.

The new approach outperforms the original one in all the cases. The overall improvement achieved with the split approach becomes larger when the number of migrating processes per node grows, since the contribution of the *WriteCkpt* and *TransferRead* phases in the original approach also increases. Particularly significant is the case of FT, because in the original version the checkpoint files have to be written to disk due to insufficient memory.

5.2.2. Impact of the chunk size

The impact of the chunk size in the *WriteTransferRead* time is analyzed in this section. Figure 10 shows the *WriteTransferRead* times for each benchmark

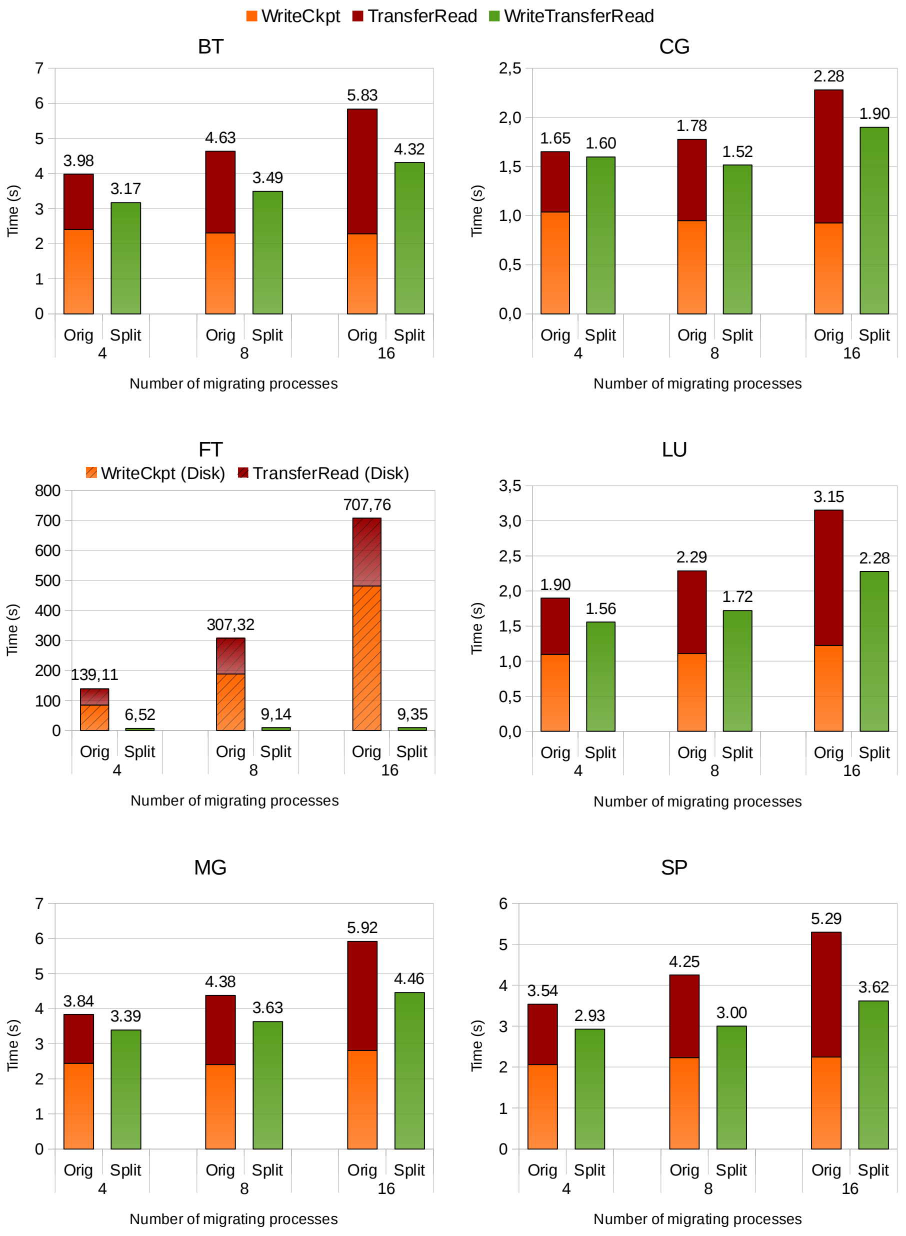

Figure 9: Times of the write-transfer-read steps in the original and the split CPPC version in a 32-process execution when 4, 8 and 16 processes are migrated.

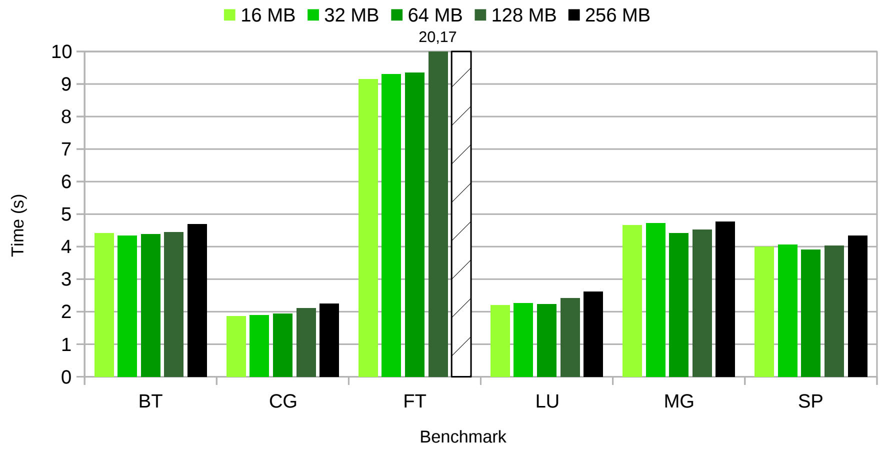

Figure 10: Times of the *WriteTransferRead* step of the split CPPC version for different chunk sizes.

running 32 processes and migrating 16 processes from the same node, for chunk sizes from 16 to 256 MB. The times for FT with a chunk size of 256 MB are not shown as there is not enough memory to store that chunk size.

The largest improvement have been obtained mainly using intermediate chunk sizes, being 64 MB (the default value) the optimal value in most of the experiments. For smaller chunks the migration time is higher because more messages are needed, and, thus, the transfer overhead is bigger. On the other hand, when the chunks are larger, the overlap between writing and read is smaller, since the writing of the first chunk and the read of the last one cannot be overlapped. Regardless, the sum of the *WriteCkpt* and the *TransferRead* in the original CPPP approach is always larger than the *WriteTransferRead* times of the pipeline version for all the chunk sizes considered. Thus, the new proposal reduces the write-transfer-read times in all the cases.

6. Conclusion

Checkpoint-based migration is the most popular technique to perform migration at the process level. It makes use of state files stored in memory and transferred to new locations, and restart mechanisms to perform the migration operation. However, the main drawback of checkpoint-based migration is its high overhead, both in terms of memory consumption, to store the state files, and in terms of high I/O cost, to transfer them. To overcome this issue, this work proposes to split the checkpoint files of an application-level migration approach into multiple smaller files, in order to overlap the writing of the state in the terminating processes with the read and restarting operation in the newly spawned processes. The solution proposed uses the HDF5 library, and respects the hierarchy of checkpoint files used in the original version, preserving the most important feature of the CPPC framework, the portability.

Experimental results prove the efficiency of the solution, both in terms of reduction in memory consumption and I/O migration times. The reduction in migration time is always worthwhile, but it becomes of particular importance in solutions where the migration of processes are used to prevent application failures proactively, i.e., the processes are migrated away from those nodes that are supposed to fail. Also, the reduction in memory consumption, when the application presents very large checkpoint files, allows for avoiding the use of disk storage, that would inflict a significant penalty in the migration time. Proposals like this one, that aims to reduce the overhead of the migration operation, can make a difference when determining whether the migration operation is viable or not.

Acknowledgment

This work has been supported by the Ministry of Economy and Competitiviness of Spain a(project TIN2013-42148-P) and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Units (Ref. GRC2013/055), both confunded by FEDER funds of the EU.

References

[1] M. Rodríguez, I. Cores, P. González, M. J. Martín, Improving an MPI application-level migration approach through checkpoint file splitting, in: Proceedings of the 26th IEEE International Symposium on Computer Architecture and High Performance Computing, Paris, France, 2014, pp. 33–41.

[2] I. Cores, G. Rodríguez, M. J. Martín, P. González, In-memory application-level checkpoint-based migration for MPI programs, The Journal of Supercomputing (2014) 1–11.
URL `http://dx.doi.org/10.1007/s11227-014-1120-2`

[3] G. Rodríguez, M. J. Martín, P. González, J. Touriño and R. Doallo, CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications, Concurrency and Computation: Practice and Experience 22 (6) (2010) 749–766.

[4] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and migration of UNIX processes in the Condor distributed processing system, Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department (April 1997).

[5] J. Cao, Y. Li, M. Guo, Process migration for MPI applications based on coordinated checkpoint, in: Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on, Vol. 1, IEEE, 2005, pp. 306–312.

[6] T. J. Hacker, F. Romero, J. J. Nielsen, Secure live migration of parallel applications using container-based virtual machines, International Journal of Space-Based and Situated Computing 2 (1) (2012) 45–57.

[7] L. V. Kale, S. Krishnan, Charm++: Parallel Programming with Message-Driven Objects, in: G. V. Wilson, P. Lu (Eds.), Parallel Programming using C++, MIT Press, 1996, pp. 175–213.

[8] C. Huang, O. Lawlor, L. V. Kalé, Adaptive MPI, in: In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03, 2003, pp. 306–322.

[9] Chakravorty, S., Mendes, C.L. and Kalé, L.V., Proactive fault tolerance in MPI applications via task migration, in: High Performance Computing-HiPC 2006, Springer, Bangalore, India, 2006, pp. 485–496.

[10] A. B. Nagarajan, F. Mueller, C. Engelmann, S. L. Scott, Proactive fault tolerance for HPC with Xen virtualization., in: Proceedings of the International Conference of Supercomputing (ICS'07), 2007, pp. 23–32.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), 2003, pp. 164–177.

[12] C. Engelmann, G. R. Vallee, T. Naughton, S. L. Scott, Proactive fault tolerance using preemptive migration, in: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009, pp. 252–257.

[13] R. Singh, P. Graham, Performance driven partial checkpoint/migrate for lam-mpi, in: High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on, IEEE, 2008, pp. 110–116.

[14] Berkeley Lab Checkpoint/Restart, last accessed July 2015.
URL `https://ftg.lbl.gov/checkpoint/`

[15] Wang, C., Mueller, F., Engelmann, C. and Scott, S.L., Proactive process-level live migration in HPC environments, J. Parallel Distrib. Comput 72 (2012) 254–267.

[16] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, last accessed July 2015.
URL `http://mvapich.cse.ohio-state.edu/`

[17] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, D. K. Panda, RDMA-based job migration framework for MPI over infiniband, in: Cluster Computing (CLUSTER), 2010 IEEE International Conference on, IEEE, 2010, pp. 116–125.

[18] C. Du, X.-H. Sun, MPI-Mitten: Enabling migration technology in MPI, in: Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on, Vol. 1, IEEE, 2006, pp. 11–18.

[19] C. Du, X.-H. Sun, K. Chanchio, HPCM: A pre-compiler aided middleware for the mobility of legacy code, in: CLUSTER, 2003, pp. 180–.

[20] C. J. Li, W. K. Fuchs, CATCH: Compiler-Assisted Techniques for Checkpointing, in: 20th International Symposium on Fault Tolerant Computing (FTCS-20), 1990, pp. 74–81.

[21] J. S. Plank, K. Li, Ickp: A consistent checkpointer for multicomputers, IEEE Parallel Distrib. Technol. 2 (2) (1994) 62–67.

[22] J. Plank, M. Beck, G. Kingsley, Compiler-assisted memory exclusion for fast checkpointing, IEEE Technical Committee on Operating Systems and Application Environments 7 (4) (1995) 10–14.

[23] X. Ouyang, R. Rajachandrasekar, X. Besseron, D. K. Panda, High performance pipelined process migration with RDMA, in: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE Computer Society, 2011, pp. 314–323.

[24] I. Cores, G. Rodríguez, M. J. Martín, P. González, R. R. Osorio, Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes, New Generation Computing 31 (3) (2013) 163–185. `doi:10.1007/s00354-013-0302-4`.

[25] G. Rodrıguez, M. J. Martın, P. González, J. Tourino, A heuristic approach for the automatic insertion of checkpoints in message-passing codes, Journal of Universal Computer Science 15 (14) (2009) 2894–2911.

[26] G. Rodríguez, M. J. Martín, P. González, J. Touriño, Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study, The Computer Journal 54 (11) (2011) 1821–1837.

[27] HDF-5 Hierarchical Data Format, last accessed July 2015.
URL `http://www.hdfgroup.org/HDF5/`

[28] The NAS Parallel Benchmarks, last accessed July 2015.
URL `http://www.nas.nasa.gov/publications/npb.html`