Failure avoidance in MPI applications using an application-level approach

IVÁN CORES, GABRIEL RODRÍGUEZ, PATRICIA GONZÁLEZ, MARÍA J. MARTÍN

Computer Architecture Group, University of A Coruña, Spain
Email: {ivan.coresg,grodriguez,pglez,mariam}@udc.es

Execution times of large-scale computational science and engineering parallel applications are usually longer than the mean-time-between-failures (MTBF). For this reason, hardware failures must be tolerated by the applications to ensure that not all computation done is lost on machine failures. Checkpointing and rollback recovery is one of the most popular techniques to provide fault tolerance support to parallel applications. However, when a failure occurs, most checkpointing mechanisms require a complete restart of the parallel application from the last checkpoint. New advances in prediction of hardware failures have led to the development of proactive process migration approaches, where tasks are migrated in a preventive way when node failures are anticipated, avoiding the restart of the whole application. The work presented in this paper extends an application level checkpointing framework to proactively migrate MPI processes when impending failures are notified, without having to restart the entire application. The main features of the proposed solution are: low overhead in failure-free executions, avoiding the checkpoint dumping associated to rolling back strategies; low overhead at migration time, by means of the design of a light and asynchronous protocol to achieve a consistent global state; transparency for the user, thanks to the use of a compiler tool and a runtime library; and portability, since it is not locked into a particular architecture, operating system or MPI implementation.

Keywords: Failure Avoidance, Proactive Migration, Checkpointing, Message-Passing

1. INTRODUCTION

The current trend in computer architecture is the use of large clusters, often heterogeneous, in which the nodes are multi/many-core systems. These are highly dynamic systems, with an everincreasing number of processors, which causes relatively high hardware failure rates [1]. For parallel programs executing on a large number of processors, this translates into frequent execution failures and a decrease in productivity.

Many fault tolerance methods for parallel applications on clusters exist in the literature, checkpoint and rollback recovery [2] being the most popular. It periodically saves the computation state to stable storage, so that the application execution can be resumed by restoring such state. In case of failure, most of the current checkpointing and rollback solutions restart all the processes from their last checkpoint. However, a complete restart is unnecessary, since most of the nodes will still be alive. Moreover, it has important drawbacks. First, full restart implies a job requeueing, with the consequent loss of time. Second, since the assigned set of execution nodes is, in the general case, different from the original one, checkpoint data must be moved across the cluster in order to restart the computation, usually causing significant network contention and therefore high overheads. These limitations can be overcome if affected processes may be individually restarted in case of a single node failure [3].

The aforementioned approaches use the checkpoint files to respond in the event of a failure. However, with the recent advances in monitoring systems, and thus in the prediction of hardware failures [4], solutions that use checkpointing to implement *proactive* policies have emerged [5, 6]. In these approaches tasks are preemptively migrated from processors that are about to fail. Thus, only terminating processes need to dump their state, reducing the usually high I/O overhead associated to checkpointing solutions. Studies show failure avoidance to be more efficient than traditional fault tolerance [7]. Moreover, both techniques can complement each other, reducing checkpoint frequency when the success rate of failure prediction is high [6].

This paper presents a checkpoint-based proposal to implement failure avoidance using proactive process migration in MPI codes. The work extends the CPPC framework [8], a portable and transparent checkpointing infrastructure for parallel applications, to proactively migrate processes when impending failures are notified. The work makes the following major contributions:

- a light and asynchronous protocol to achieve a global consistent state during the migration operation. This protocol avoids rollback and its associated loss of already done computation.
- lightweight checkpointing and migration techniques; the read/write operations are overlapped with computation or other migration-related operations whenever

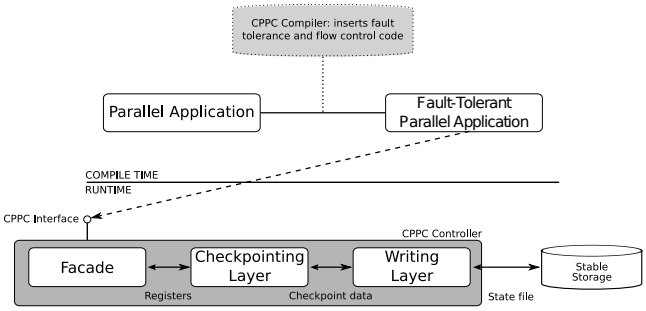


FIGURE 1. Integration of a parallel application with the CPPC framework

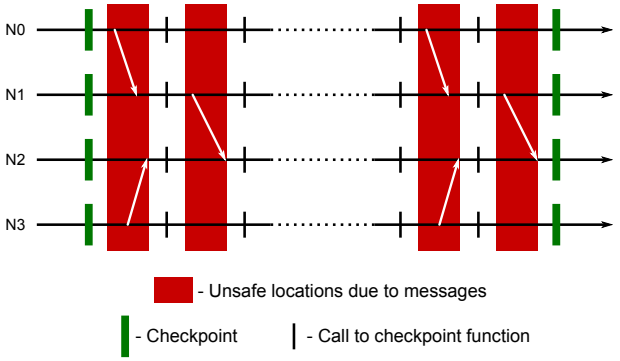


FIGURE 2. Spatial coordination for checkpointing

possible.

- an application-level migration solution. It does not make any assumptions about the underlying system hardware/software characteristics (including the MPI implementation), thus enabling portable operation.

The structure of this paper is as follows. Section 2 presents an overview of CPPC, the checkpointing tool extended in this work. Issues related to proactive process migration using CPPC and its implementation are discussed in Section 3. Section 4 evaluates the performance of the proposed solution, demonstrating its efficiency and feasibility. Section 5 describes related work. Finally, Section 6 concludes the paper and discusses future work.

2. THE CPPC FRAMEWORK

The proactive process migration mechanism proposed in this paper has been implemented on top of CPPC [8–10], an application level checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications. CPPC appears to the user as a compiler tool and a runtime library. The integration between the application and the CPPC framework is automatically performed by the CPPC compiler, that translates the application source files into derived files with added checkpointing capabilities. The global process is depicted in Figure 1. At compile time, the CPPC compiler is used to automatically transform a parallel application into a fault-tolerant parallel application with calls to the CPPC library. The following subsections detail relevant design aspects of CPPC.

2.1. The CPPC approach

There are several issues to be solved in implementing practical checkpoint solutions for parallel applications, such as checkpoint consistency, memory requirements and portability.

As for checkpoint consistency, the basic difference between sequential and parallel applications is the existence of dependencies imposed by interprocess communications. If a checkpoint is placed in the code between two matching communication statements, an inconsistency would occur upon recovery, since the first one will not be executed. Several solutions have been proposed to ensure the consistency of a checkpointing scheme [2] being coordinated approaches the most common practical choice [11, 12], due to their recovery process being very simple. However, an important drawback of classical coordinated protocols is their lack of scalability [13]. CPPC minimizes the runtime overhead of classical consistency protocols by using a non-blocking spatially coordinated approach [9]: checkpoints are taken at the same relative code locations by all processes (an SPMD programming model is assumed), but not forcibly at the same time. The proposal implies identifying, at compile time, code locations where it is guaranteed that no inconsistencies due to messages may occur (see Figure 2). These code locations are called safe points. To automatically identify safe points, the compiler performs a static analysis of inter-process communication. Afterward, a heuristic analysis, based on code complexity, selects the best safe points for checkpoint file dumping, inserting a checkpoint function (`CPPC_Do_checkpoint()`) there. However, not all checkpoint function calls will generate checkpoint files. During runtime a checkpoint frequency may be defined in terms of number of calls to the checkpoint function. Each time this function is called an internal CPPC parameter, `touchedCheckpoints`, will be increased. This parameter will be used to number the different checkpoint files. During restart the application processes perform a negotiation phase to identify the most recent valid recovery line, formed by the newest checkpoint file available simultaneously to all processes. By statically ensuring that checkpoints may occur only at selected safe locations, no interprocess communications or runtime synchronization are necessary. In this way, the static coordination protocol achieves consistency by using compile-time introduced constraints, improving efficiency and scalability by transferring consistency-related actions from runtime to both compile and restart time.

Regarding memory requirements, CPPC works at the variable level (i.e. storing user variables only) and performs a live variable analysis that identifies which variable values are needed for the correct restart of the execution. Live variables are automatically detected by the CPPC compiler and marked using a CPPC function (`CPPC_Register()`)

```
int main(int argsc,
         char ** argsv) {

   // Variable definitions
   ...

   MPI_Init(&argsc, &argsv);

   // Matrix data input and distribution
   ...

   for( i=0; i < niters; i++ ) {
      // Matrix diagonalization
      ...
   }

   ...

   MPI_Finalize();
}
```

FIGURE 3. Skeleton of an example of MPI code: a matrix diagonalization

```
int main(int argsc,char ** argsv){

   // Variable definitions
   ...

   MPI_Init( &argsc, &argsv );

   CPPC_Init( &argsc, &argsv );
   // Conditional jump to CPPC_EXEC_1
   if( CPPC_Jump_next() ) {
      goto CPPC_EXEC_1;
   }

   // Matrix data input and distribution
      ...

CPPC_EXEC_1:
   CPPC_Register(&i, ...  );
   ...
   //Conditional jump to CPPC_EXEC_2

   for(i=0; i < niters; i++){
CPPC_EXEC_2:
      CPPC_Do_checkpoint( 0 );
      //Conditional jump to CPPC_EXEC_3

      // Matrix diagonalization
      ...
   }

   ...
   CPPC_Shutdown();
}
```

FIGURE 4. CPPC-instrumented matrix diagonalization example code

to provide such information to the CPPC controller. This process is referred to as "variable registration". Besides, CPPC applies other snapshot size reduction technique, zero-blocks exclusion [14], which consists in avoiding the storage of memory blocks that contain only zeros. Working at the variable level allows both to reduce the amount of data to be saved, which is one of the most performance impacting factors in checkpointing, and to store only portable data, hence making restart possible on different architectures. Using liveness information and zero-blocks exclusion further reduces snapshot sizes. Storing only portable data on state files introduces the need for some kind of recovery mechanism, capable of regenerating the nonportable state that is not stored into state files. This mechanism is further described in the next subsection.

2.2. CPPC Operation

For illustrative purposes, Figure 3 shows the C code of an MPI application (a matrix diagonalization) and Figure 4 details the fault-tolerant version of the same code obtained by using the CPPC source-to-source compiler.

CPPC has two operation modes: *checkpoint* and *restart*. Checkpoint mode is used during regular execution. Processes execute the code sequentially and create checkpoints according to their specified checkpoint frequency. Restart mode is used after the original execution has aborted to recover the computation state of all processes from a previously saved snapshot. CPPC uses code re-execution to recover the application state. A section of code is defined as Required-Execution Code (REC) if it must be re-executed during a process restart to ensure correct state recovery. Each REC recovers some part of the original application state. The fundamental REC types are nonportable calls, variable registrations, and checkpoint calls. A typical example of a nonportable call is a call to a function manipulating opaque library state, such as an MPI function which creates or modifies a communicator.

The CPPC compiler divides applications into pieces formed by: a block of non-relevant code, a jump target (`CPPC_EXEC` labels in the figure), a block of restart-relevant code (REC), and a conditional jump to the next jump target, which will be placed right before the following REC. Conditional jumps will only be taken when in restart mode. In this way, after a failure, CPPC is able to re-execute only relevant parts of the code, skipping the non-relevant ones.

Following the example, during *checkpoint* operation the MPI environment is initialized, then the CPPC controller; matrix data are read and distributed; relevant variables are registered by every process (loop index, loop limit and matrix data); next the core computation of the application begins with calls to the checkpoint function in every iteration and actual checkpoint dumping every n iterations depending on the specified checkpoint frequency; and, finally, the results are written and CPPC is shut down.

In *restart* operation the execution starts normally. Upon calling the CPPC initialization function the restart is detected, and a negotiation phase is performed to identify the most recent recovery line, that is, the set of checkpoint files to be used for restart. These files are verified and read, and restart mode is entered, which activates the conditional jumps that direct the execution through the identified RECs and skip nonrelevant sections of code. In the example, the matrix data input will be skipped. The variable registration

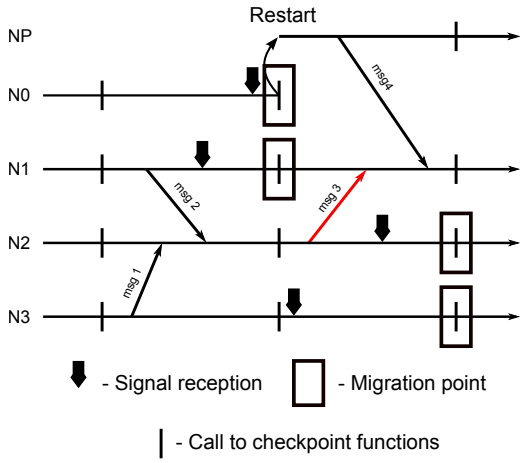


FIGURE 5. Inconsistent global state after migration in processes that are running asynchronously

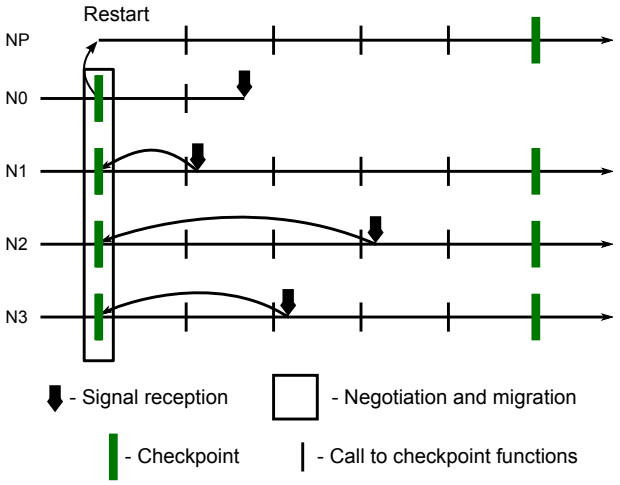


FIGURE 6. Backward negotiation

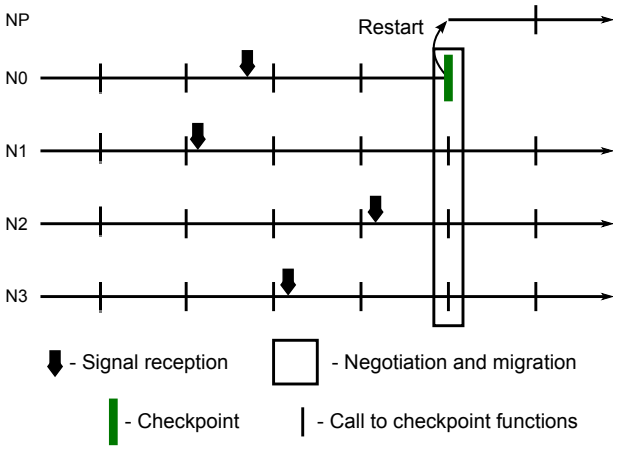


FIGURE 7. Forward negotiation

REC recovers variable values. Finally, the execution reaches the checkpoint inside the computational loop, the library is reconfigured to checkpoint mode and the application continues regular execution.

3. APPLICATION-LEVEL MIGRATION

The basic idea behind dynamic migration in parallel applications is to spawn new processes that will be in charge of continuing the work of the terminating processes on other computation nodes. Migration is preferably performed to spare nodes, although the use of already allocated ones is also possible. In a checkpoint-based solution, when a signal with a migration request is received, the terminating processes need to write their state to checkpoint files, while newly spawned processes need to read these files and recover the state of the terminating processes. Besides, before resuming the execution, 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. A possible solution to this problem is to make 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, thus facilitating the implementation of this approach. Besides, based in a heuristic evaluation of computational cost, it places calls to the 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, since messages may be sent in the code executed in between the two calls. The communication labeled *msg. 3* in Figure 5 is an example of such a situation. In order to implement proactive process migration processes need to dynamically engage in a negotiation to decide which checkpoint call to select as migration point.

Summarizing, there are two main phases on process migration using CPPC: a negotiation to reach consensus on the migration point; and the process migration itself, which includes the communicator reconstruction.

3.1. Negotiation protocol

The negotiation protocol must ensure that, when a migration is initiated, all processes are able to converge to a single selected checkpoint location to achieve global coordination. There are different approaches that can be used towards this end. A first possibility is *backward negotiation*, shown in Figure 6. Using this strategy all processes agree to restart their execution from the most recent recovery line [15]. Another solution is *forward negotiation*, detailed in Figure 7, in which processes agree to coordinate at the next checkpoint call to be reached by the process that has advanced the farthest in the execution. Backward negotiation uses previously created checkpoint files. As such, its greatest advantage is avoiding the overhead of

creating new snapshots during migration. This, in turn, incurs higher overheads during a failure-free execution, given that processes need to checkpoint often. Backward negotiation can be thought of as being roughly equivalent to a stop and restart approach but avoiding the job requeueing. All processes need to recover a previous state, causing a loss of computation and higher total execution overhead. Due to these shortcomings of backward negotiation, forward negotiation will be the approach followed in this work.

We are assuming that the `mpirun` process receives a migration request from a user or batch scheduler and propagates it by sending a signal to each of the spawned MPI processes. This external signal triggers a handler which activates a migration flag in the CPPC controller to change to *migration* mode, a new operation mode added to CPPC besides checkpoint and restart, explained in Section 2.2. In migration mode each MPI process has to coordinate with the others to find out the farthest checkpoint location that has been reached by any of them. As the `touchedCheckpoint` CPPC parameter stores the number of times this function is called for each process, a direct and simple solution is to use an MPI reduction operation inside the signal handler to calculate the maximum `touchedCheckpoint` value. Unfortunately, according to the MPI standard, implementations may prohibit the use of MPI calls from within signal handlers. Thus, for the sake of robustness and portability an alternative negotiation protocol was built outside of the signal handler.

One-sided MPI communications are used so that processes may continue running asynchronously during the negotiation. Prior to invoking a one-sided MPI communication operation, each process has to specify the memory region (window) that it exposes to others. The window for the proposed negotiation algorithm comprises two values for each process: `flag` and `touched`. The `flag` value indicates whether a process is actively engaged in the negotiation. It is activated when a checkpoint function is reached after migration mode has been enabled. It will not be deactivated until the migration is finished. The `touched` value is kept up to date throughout the execution when in checkpoint mode, to contain the value of the `touchedCheckpoints` parameter. When in migration mode this value is not updated, and contains the value it had when the migration mode was enabled by the external signal.

Algorithm 1 shows the pseudocode of the negotiation algorithm. This code is included inside the checkpoint function and executed only when in migration mode. Each process p reads the exposed `flag` and `touched` values of every other process q (`GetRemoteWindow()` in the figure). In this way, all processes have a global picture of the execution status. As explained before, each process must advance up to the farthest reached checkpoint location. If process q is more advanced than process p (i.e. $\texttt{touched}_q > \texttt{touched}_p$), then p must continue its execution until the next checkpoint location, regardless of the value of $\texttt{flag}_q$. Otherwise, a deadlock would occur if process q were waiting for a message from process p sent in the application code in between the checkpoint call number $touched_p$ and the one number $touched_q$ (hence unable to reach the next checkpoint location and activate its flag). If process q has not yet advanced beyond checkpoint $touched_p$, then p waits for q to enable its `flag` value. This indicates that q is aware that a migration is to take place. Once all processes are verified to be aware of the negotiation process and not more advanced than process p a consensus migration point has been discovered, and process p has arrived at it. Note that other processes may be behind in their execution, and will arrive later at the same migration location in an asynchronous way.

```
value[2];
for allRemoteProcesses do
    flag = 0;
    while !flag do
        LockWindow();
        GetRemoteWindow(&value);
        UnlockWindow();
        if value.touched > touchedCheckpoints then
            return; %continue to next checkpoint
        end
        flag = value.flag;
    end
end
```

Algorithm 1: Pseudocode for the negotiation protocol

Local updates to the window values use exclusive locks (`MPI_LOCK_EXCLUSIVE`) to guarantee consistency, whereas remote reads (Get operations on Figure 1) use shared locks (`MPI_LOCK_SHARED`), which allow for concurrent read accesses.

3.2. Process migration

At this point, a migration spot has been agreed upon and processes begin arriving at that location independently. Still, several issues remain to be solved: saving the terminating processes state; spawning new processes to continue the computation done by the terminating ones; updating and managing the communication groups; and restoring the terminating processes state in the newly spawned processes. Whenever possible, these actions will be taken by each process without coordination. All the required steps are fully explained below.

The state of the terminating processes is saved using native CPPC capabilities. Checkpoint file creation begins once the terminating process reaches the migration point. Note that, due to the spatial coordination protocol employed by CPPC, there is no need to coordinate processes at the migration point before the state dump can start. Checkpoint creation is managed by a new ad-hoc thread, which allows for the reconfiguration to occur concurrently.

The newly spawned processes are created using the `MPI_Comm_spawn_multiple()` MPI-2 function. This call is collective over the communicator, that is, it must be performed by all the processes in the communication

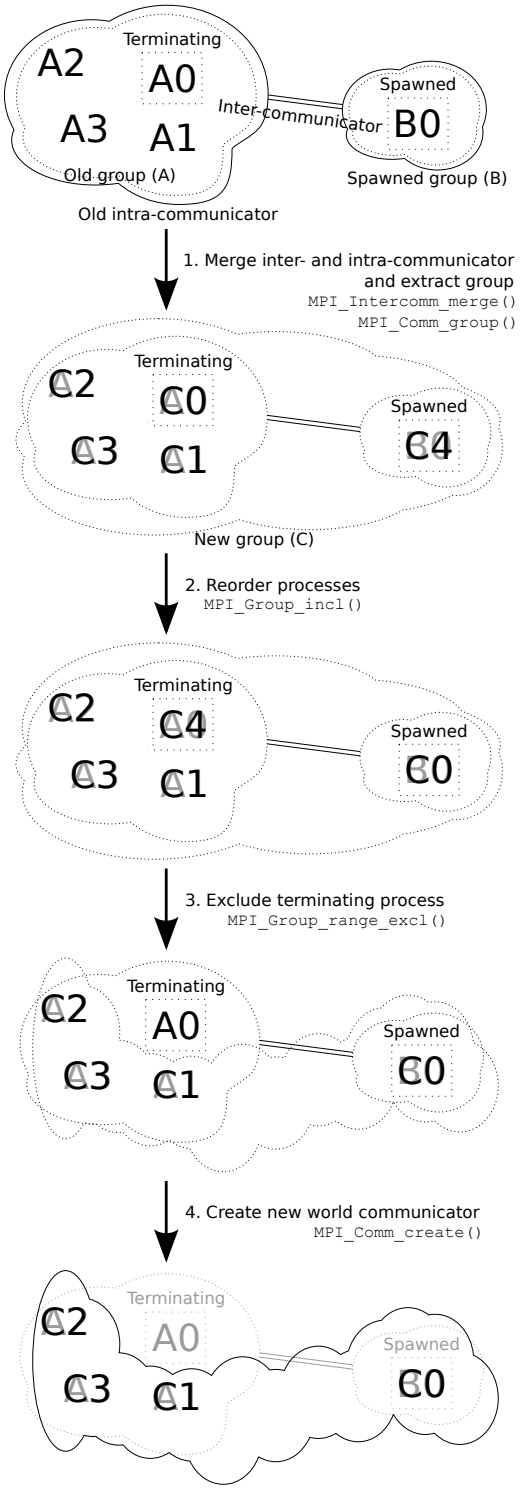


FIGURE 8. World communicator reconfiguration. Process A0, in the world communicator A, migrates to a new execution node. B0 is the newly created process to support the migration. The old world communicator A is reconfigured into a new world communicator C

group involved in the migration (that is, the world communicator). Depending on the implementation, `MPI_Comm_spawn_multiple()` may not return until `MPI_Init()` has been called in the spawned processes. Similarly, `MPI_Init()` in the spawned processes may not return until all processes in the original communicator have called `MPI_Comm_spawn_multiple()`. As such, `MPI_Comm_spawn_multiple()` in the original processes and `MPI_Init()` in the spawned ones form a collective operation over the union of parent and child processes that may imply a synchronization during the migration operation.

Spawning new processes creates an inter-communicator between the original and the newly created processes. Old communicators should be reconstructed, replacing terminating processes with the newly created ones. The approach used is to reconfigure the world communicator (`MPI_COMM_WORLD`) using the dynamic communicator management facilities provided in MPI-2. Other communicators, which derive from `MPI_COMM_WORLD`, will be reconstructed by re-executing the MPI calls used for creating them in the original execution. Figure 8 details the reconfiguration phase for the world communicator for an example where four processes take part of the migration operation and only one process is migrated to a new execution node. First, the two intra-communicators that contain the original and the new processes need to be merged into a single one. The `MPI_Intercomm_merge()` function is used for this purpose. Afterward, the group of processes that form the new intra-communicator is extracted via `MPI_Comm_group()`. Ranks in this group are reordered using `MPI_Group_incl()`, in such a way that the spawned processes will take over the ranks of the terminating processes. Afterward, terminating processes are excluded from the group using `MPI_Group_range_excl()`. Finally, `MPI_Comm_create()` is used to build the new world communicator from the reconfigured group.

As described, this process only reconfigures the world communicator. However, in order for the migration to succeed, communicators which include any migrating process have to be rebuilt as well. Using the same approach for reconfiguring these communicators would require the participation of the terminating processes, which would in turn require the soon-to-fail nodes to be up for a longer time, reducing the chances of successful migration. In order to avoid this, the CPPC restart capabilities are used. MPI calls that result in the creation of new communicators (such as split operations) are identified and logged by CPPC both into memory and created checkpoint files. The set of communicators in an MPI application can be seen as a tree in which each node is created from another one by using a certain MPI operation (i.e. `MPI_Comm_split()`, `MPI_Comm_dup()`, etc.). The root of this tree is the world communicator. Taking advantage of the operation log provided by CPPC, this communicator tree is reconstructed from its root by orderly re-executing the logged operations. Regular processes (those that do not migrate) read the log from memory and re-execute its contents right after the reconfiguration of the world communicator. Spawned processes do so after reading the checkpoint file contents during their restart phase. Note that if these MPI operations are blocking, a synchronization between the processes involved will be imposed.

Terminating processes, in turn, 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 (using the inter-communicator created by `MPI_Comm_spawn_multiple()`) that checkpoints may now be read (assuming a shared file system). Finally, they safely finish their execution.

Spawned processes still have to recover the terminating processes state from their snapshots contents. This involves 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. EXPERIMENTAL RESULTS

Experiments were performed to evaluate both the scalability of the proposed solution and the total overhead associated to the migration. A multicore cluster was used to carry out these experiments. It consists of 8 nodes powered by two quad-core Intel Xeon E5620 CPUs with 16 GB of RAM. The cluster nodes are connected through an Infiniband network. The front-end is powered by one quad-core Intel Xeon E5502 CPU with 4 GB of RAM. The connection between the front-end and the execution nodes is an Infiniband network too. The working directory is mounted via NFS and is connected to the cluster by a Gigabit Ethernet network. All the checkpoint files were stored into this working directory.

The application testbed was comprised of six out of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [16] (NPB from now on) compiled with the OpenMPI library version 1.5.4. The IS and MG benchmarks were discarded due to their low execution times.

The experimental results obtained are classified in two subsections. The first one evaluates the scalability of the solution, analyzing the duration of the different phases of a migration operation in relation to different impacting factors. The second subsection evaluates the migration overhead of the proposed approach, and compares it with other different solutions.

4.1. Scalability

The scalability of the solution can be analyzed from three points of view: the impact of the total number of processes in the execution, the effect of the number of migrating processes, and the influence of the application memory footprint. In all these experiments the migration time is broken down into 5 parts (see Figure 9):

- *Negotiation*: execution time between the `mpirun` migration request and the call to the spawn function. This time is measured in the worst possible case, that is, when the signal is received by at least one of the processes just after a checkpoint function call.
- *Spawn&Rec*: execution time of the spawn function and the reconfiguration of the world communicator.

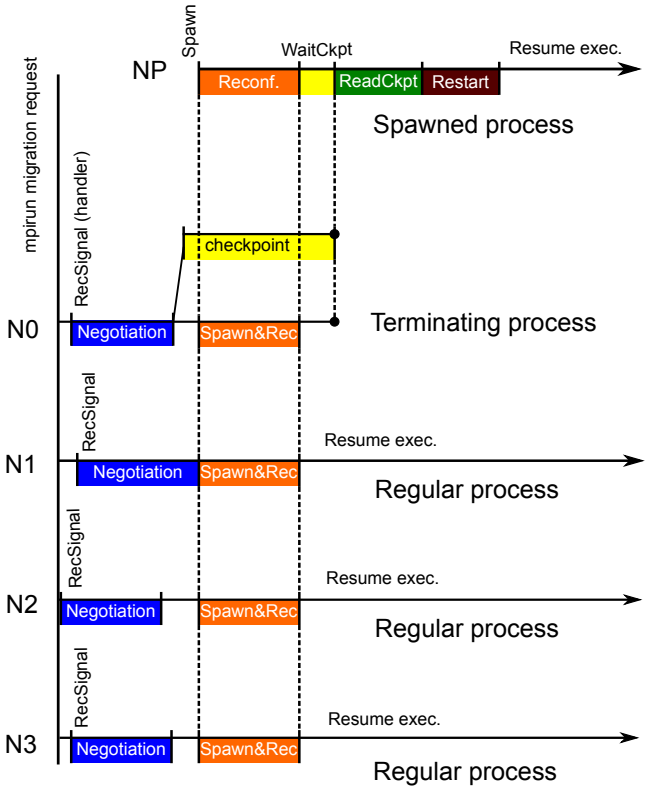


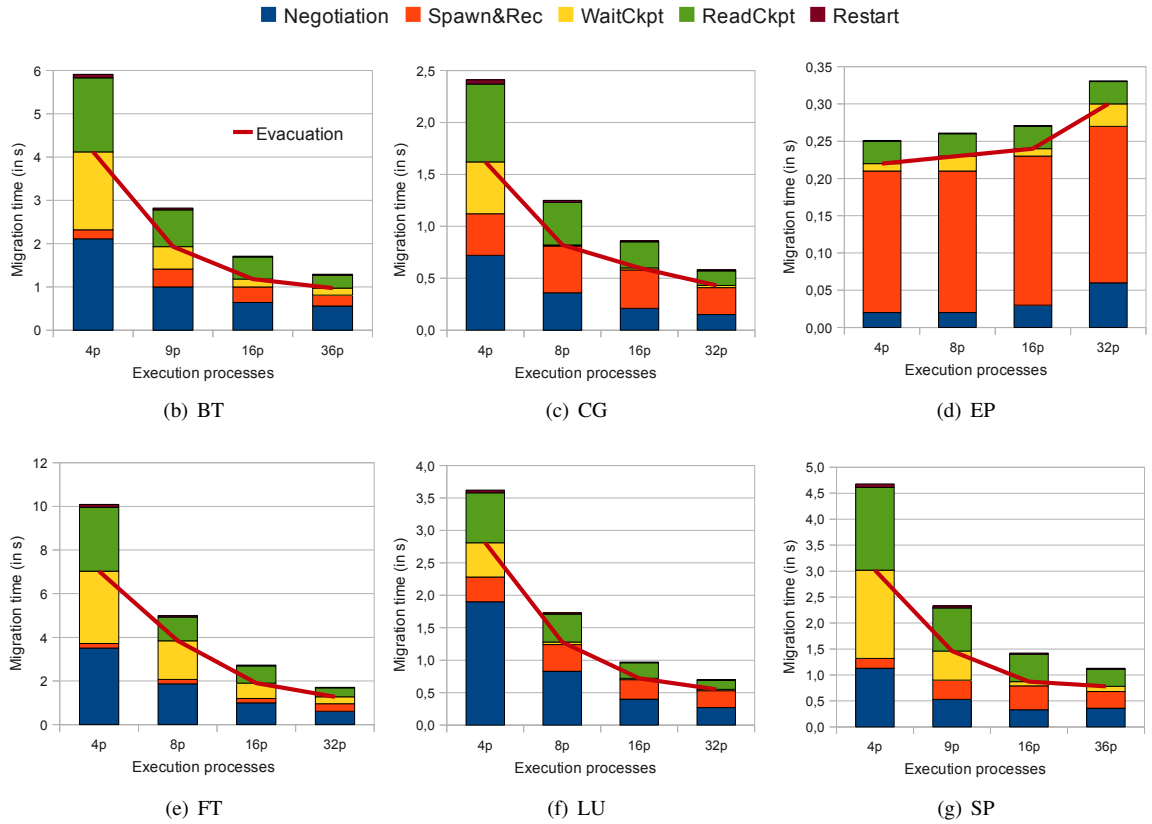
FIGURE 9. Actions and temporal sequence for four processes involved in a migration operation. Process $N0$ migrates to a new execution node. NP is a newly created process to support the migration. $N1-N3$ are regular processes that passively participate in the migration

- *WaitCkpt*: average execution time between the end of the reconfiguration phase in the newly spawned processes, and the start of the checkpoint file read.
- *ReadCkpt*: average time it takes to read the checkpoint file from disk in the newly spawned processes.
- *Restart*: average time for restarting the application once the checkpoint file has been read. It includes the execution of the RECs.

4.1.1. Impact of the number of processes

These tests measure the scalability of the migration solution when increasing the number of total processes. Experiments were carried out using 4, 8, 16 and 32 processes, except for BT and SP that need a square number of processes, thus using 4, 9, 16 and 36. Although each node was running at least 4 processes, in this experiment only one process was migrated each time (equivalent to what would occur if an imminent failure was predicted in one node with one process running on it). In all the cases the terminating process was migrated to a spare node. The results obtained using the NPB applications using class B are shown in Figure 10.

The *Negotiation* time depends on how often the CPPC checkpoint function is called, the inherent synchronization between the processes during the execution of the application, and the overhead introduced by the negotiation pro-



(b) BT (c) CG (d) EP

(e) FT (f) LU (g) SP

FIGURE 10. Scalability impact when increasing the number of total processes. NPB class B migrating one process

TABLE 1. Negotiation times and iteration times, running 16 processes

NPB. Class B	*Negotiation (s)*	*Iteration time (s)*
BT	0.64	0.56
CG	0.21	0.20
EP	0.03	0.01
FT	1.00	1.00
LU	0.40	0.39
SP	0.33	0.28

TABLE 2. Iteration times (in s) for different number of total processes

NPB. Class B	*4p*	*8/9p*	*16p*	*32/36p*
BT	2.05	0.94	0.56	0.28
CG	0.73	0.38	0.20	0.15
EP	0.01	0.01	0.01	0.01
FT	3.59	1.85	1.00	0.61
LU	1.46	0.73	0.40	0.24
SP	1.14	0.50	0.27	0.14

tocol. Given that the migration signal is received just after a checkpoint call, the *Negotiation* time will be at least the time between two consecutive checkpoint calls. In all NPB, the CPPC checkpoint function is called once in each iteration of the main computational loop. The *Negotiation* time, as well as the execution time per iteration, for 16 processes and class B, are shown in Table 1. Except for EP, the processes of all the NPB 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 in Table 1 prove that in these cases the overhead associated to the negotiation protocol is almost negligible, being the *Negotiation* time mainly determined by the iteration time. As the number of processes increases, the iteration time decreases, thus achieving a reduction in the *Negotiation* time. Table 2 shows the execution time per iteration for different number of processors. However, when the processes are not synchronized, such as in EP, it may take several checkpoint calls to reach the migration point.

The *Spawn&Rec* time increases slightly with the total number of processes, since this phase involves different collective communications. However, as it can be observed in Figure 10, this time is at most 0.5 seconds in these experiments.

As it can be seen in the figure, in most of the cases, the biggest contribution to the migration overhead is due to the write and read of checkpoint files. In these experiments checkpoint files are stored to shared disk via NFS, using a Gigabit Ethernet network. In this situation, checkpoint file sizes are critical to minimize the I/O time. CPPC applies live variable analysis and identification of zero-blocks to decrease checkpoint file sizes [14]. These sizes, as well as the checkpoint write and read times, for NPB class B

TABLE 3. Checkpoint file size per process (in MB) and checkpoint write and read times (in s)

NPB. Class B	*Size*	*Write T.*	*Read T.*
BT	30.69	0.51	0.58
CG	14.23	0.33	0.25
EP	1.10	0.04	0.03
FT	48.10	0.87	0.79
LU	14.49	0.25	0.24
SP	30.93	0.52	0.53

TABLE 4. Checkpoint file sizes per process (in MB) for different number of total processes

NPB. Class B	*4p*	*8/9p*	*16p*	*32/36p*
BT	106.61	52.10	30.69	17.27
CG	47.48	24.79	14.23	7.64
EP	1.10	1.10	1.10	1.10
FT	192.12	96.11	48.10	24.10
LU	48.79	26.62	14.49	8.24
SP	96.35	50.19	30.93	18.24

and 16 processes, are shown in Table 3. When the number of processes grows, the checkpoint files use to become smaller and, thus, the time to write or read the file from disc decreases. Table 4 shows the checkpoint file sizes for the different number of processes. Again, the EP application is a special case, since the checkpoint file sizes does not decrease when the number of processes grows. Thus, the checkpoint write and read times for EP remains constant in Figure 10. Note that the write of the checkpoint file is overlapped with the spawn and reconfiguration phase (see Figure 9), and the time shown in these figures is the one consumed in the non overlapped part (*WaitCkpt*).

Finally, the *Restart* time is very small for all the tested applications. It depends on the amount of state recovered using code re-execution (RECs) on the newly spawned processes. As it happens for the checkpoint file sizes, by increasing the total number of processes, the amount of state to be recovered usually decreases, and so does the *Restart* time.

Note that the sum of *Negotiation*, *Spawn&Rec* and *WaitCkpt* corresponds to the *evacuation* time, that is, the time needed after the migration request to free the nodes that are about to fail. This time is represented with a line in Figure 10. Evacuation time decreases when scaling the number of processes, except for EP. Although the evacuation time in EP increases slightly with the number of processes, its variance in absolute value is lower than 0.1s. This behavior can be explained by the negative impact of the scaling in the *Negotiation* time for the EP application.

In order to make the proposed solution practical, the evacuation time should be smaller than the lead-time (time ahead of the potential occurrence of a failure) of the prediction mechanism. In all the experiments the evacuation time was only of a few seconds. In [17] lead-times between tens of seconds and several minutes are reported. Thus, the evacuation time observed in these experiments proves the viability of the solution.

4.1.2. Impact of the number of terminating processes

When a node is about to fail, all the processes running on it have to be migrated to a new location. The previous subsection shows experimental results obtained assuming only one terminating process. In this subsection experiments were carried out varying the number of terminating processes (from 1 to 8) and maintaining the number of total processes in 16. In these experiments each node runs 2 processes and the terminating processes are migrated to spare cores of nodes that are not going to fail. The results are shown in Figure 11.

As expected, the *Negotiation* time remains constant, as the execution time between two consecutive calls to the CPPC checkpoint function does not change.

The *Spawn&Rec* time increases with the number of migrating processes. It is particularly low for migrating a single process and augments significatively when going from 1 to 2 migrating processes. This is probably due to internal OpenMPI optimizations of the `MPI_Comm_spawn_multiple()` function when spawning only one process.

The checkpoint write and read times also grow because the number of checkpoint files dumped to the NFS shared directory increases. Note that, in most cases, the *WaitCkpt* time is negligible because the time needed to dump the checkpoint files is overlapped by the increase in the time spent in the *Spawn&Rec* phase.

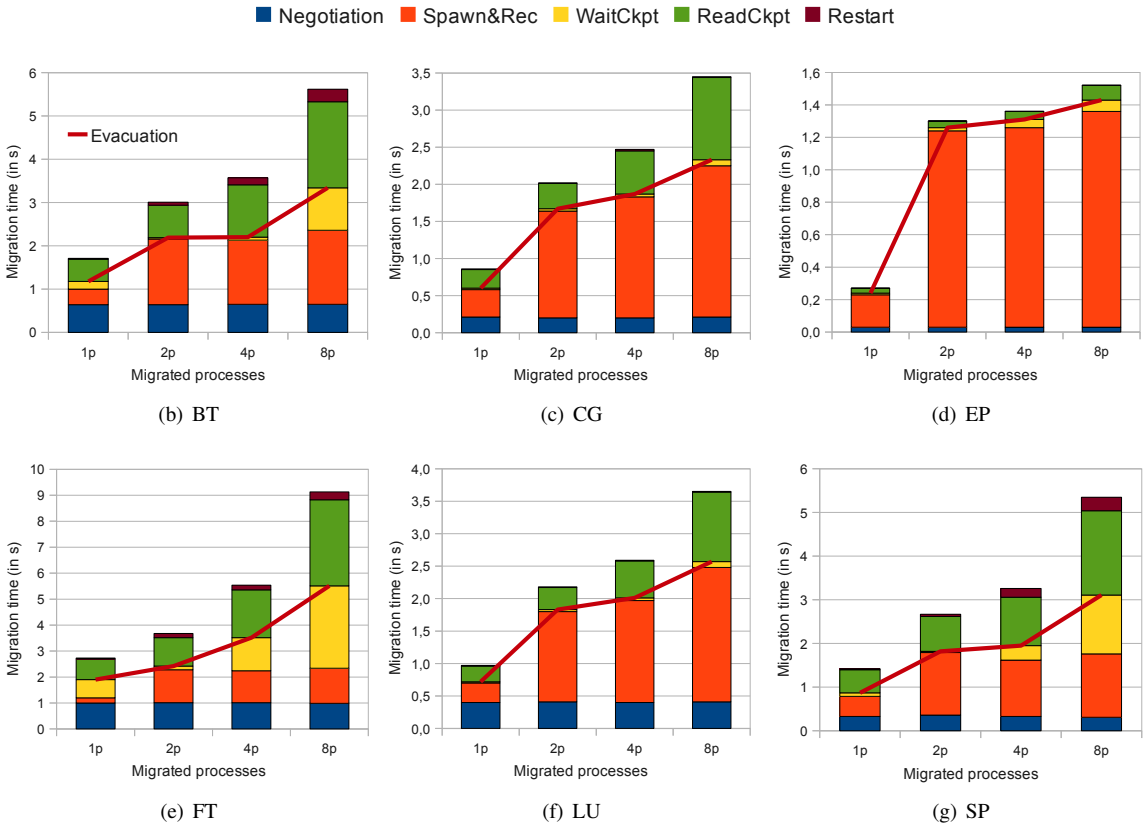
Finally, the *Restart* time increases with the number of migrating processes for those applications that recover non-world communicators during their restart phases (BT, FT, and SP). The reason for this is that, as mentioned in Section 3.2, the collective operations executed during this recovery are blocking. As such, these operations impose a synchronization that becomes more costly as the number of migrated processes increases.

4.1.3. Impact of the application size

In this subsection NPBs of classes A, B and C are used to evaluate the impact in the migration time of different application memory footprints. Figure 12 depicts the experimental results obtained when scaling the problem size, using 16 processes and migrating only one.

As expected, for most cases an increase in the migration duration is observed, since the problem scaling results in larger data per process. The *Negotiation* time grows due to the increase of the iteration time. The *Spawn&Rec* time remains almost constant, since it does not depend on the memory footprint of the application. Both the checkpoint write and read times and the *Restart* time augment due to the increase in the checkpoint file sizes and in the amount of state to be recovered, respectively.

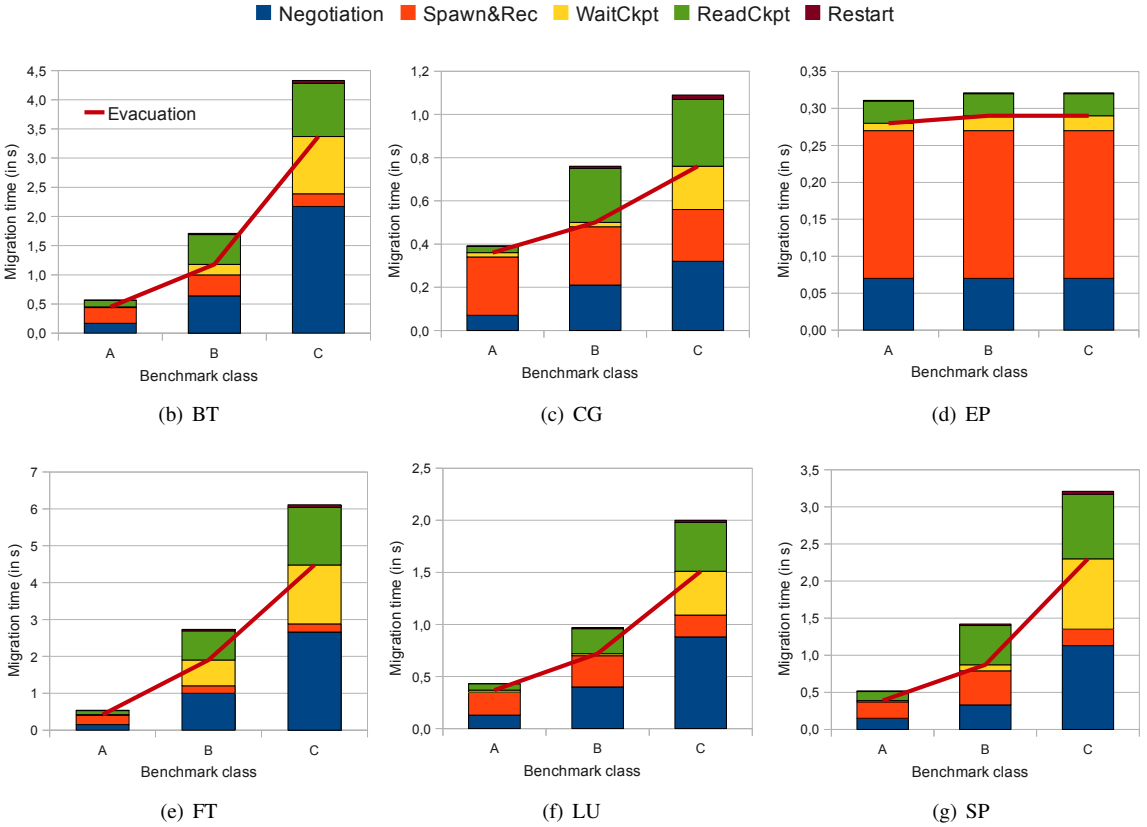
Though the problem scaling leads to an increase in the migration duration, considering the total execution time of the application, a decrease in percentage terms is observed when the problem size increases.



(b) BT (c) CG (d) EP

(e) FT (f) LU (g) SP

FIGURE 11. Scalability impact when increasing the number of migrating processes. NPB class B and 16 processes



(b) BT (c) CG (d) EP

(e) FT (f) LU (g) SP

FIGURE 12. Impact of increasing application size via the NPB classes. Running on 16 processes

TABLE 5. Execution times (in s) using 16 processes, with and without migration

	CPPC			Ckpt&	MVAPICH	
NPB. Class C	*Orig.*	*Instr.*	*Migr.*	Rollback	*Orig.*	*Migr.*
BT	449.94	450.55	459.63	482.27	471.48	486.39
CG	43.91	43.87	45.93	53.75	47.32	59.04
EP	33.27	33.29	34.59	33.70	34.35	36.01
FT	99.49	99.42	109.56	150.73	152.38	184.95
LU	418.41	419.17	425.47	433.39	435.47	444.47
SP	518.74	520.17	532.18	547.37	548.52	568.72

4.2. Overhead

The overhead was studied for each of the NPB codes by running class C with 16 processes divided in 8 nodes. In this configuration, when a node is about to fail, 2 processes are migrated. Class C was chosen to get a more realistic execution time. The results are shown in Table 5 where the column labeled *Orig.* shows the execution time of the original application in a fault-free execution and the *Instr.* one the time of the application instrumented with CPPC, again in a fault-free execution. In most cases the instrumentation overhead is minimal, generally less than 1%. The application execution times when a failure is imminent and the migration of a node is performed are shown in the *Migr.* column. As seen in the previous sections, the migration time is mainly dominated by the times to read and write the checkpoint files. Thus, FT is the application with the highest relative overhead (10% with respect to the original code) due to their larger checkpoint files (see Table 3).

Table 5 also includes the execution time of the checkpoint and rollback solution using CPPC. Checkpoint files are periodically dumped and, in case of failure, the complete execution is restarted in new nodes. The Checkpoint&Rollback times shown in this table are the optimal ones for this approach, that is, only one checkpoint file is dumped before the failure occurs, thus avoiding additional overhead due to useless dumps; and the rollback is performed just after the checkpoint, thus avoiding loss of work on restart. The overhead associated to proactive migration is lower than the overhead associated to the checkpoint and rollback solution, the only exception being EP. The relatively high overhead of the EP benchmark (1.21%) is due to the high number of MPI window updatings as a consequence of the high number of internal iterations (more than 500 iterations in approximately 5.5 seconds). Fortunately it is rather improbable to find this behavior in a real application. It can be concluded that the proactive migration approach can significantly decrease the cost to survive a node failure.

For comparative purposes, Table 5 also shows the execution time using MVAPICH version 1.8. MVAPICH provides process migration based on BLCR (Berkeley Lab's Checkpoint/Restart Library) [18] and FTB (Fault Tolerant Backplane) [19] libraries for Infiniband, iWAPP and RoCE architectures [20]. The table shows the MVAPICH original execution time, that is, a fault-free execution, and the execution time when one node needs to be migrated. Currently the MVAPICH process migration support is only available for Mellanox Infiniband adapters. Unfortunately, cluster Pluton has a QLogic card, and the Mellanox interface over the QLogic cards does not achieve its top performance, resulting in a MVAPICH fault-free execution slower than the OpenMPI execution. Regardless, the solution proposed in this paper results advantageous for all the NPB applications also in percentage terms as shown in Figure 13. This figure shows the overhead with respect to a fault-free execution. In order to provide a fair comparison, the MVAPICH overhead is calculated with respect to the MVAPICH fault-free execution. The benefit obtained using CPPC versus MVAPICH can be in part explained due to the smaller size of the checkpoint files. Table 6 shows the sizes of the checkpoint files per process generated for each application using CPPC and BLCR. Those applications where CPPC achieves significant reductions in checkpoint sizes (CG and FT) also present lower overhead under migration.

TABLE 6. Checkpoint sizes (in MB) per process (running in 16 processes) for CPPC and BLCR

NPB. Class C	*CPPC*	*BLCR*
BT	110,23	111.04
CG	27.74	72.19
EP	1.04	2.36
FT	192.12	450.36
LU	50,89	51.60
SP	103,14	100.61

In Section 4.1.1 it can be seen that EP migration time with 32 processes is negatively affected by an increase in the Negotiation time. Results in Figure 13 evidence that the coordination between processes to perform the migration operation is a bottleneck for EP, since for this application the best performance is obtained by the checkpoint and rollback approach, that avoids the coordination during execution time.

When comparing MVAPICH with the CPPC-based proposal it must be noted that one of the most important features of the latter, besides its efficiency, is its portability, as it does not need any specific architecture, operating system (OS), MPI implementation or system file to work.

Finally, results in Figure 13 also provide an idea of the impact that false-positives associated with the failure prediction may have on system performance. The overhead of migration is, for most of these benchmarks, less than 3%

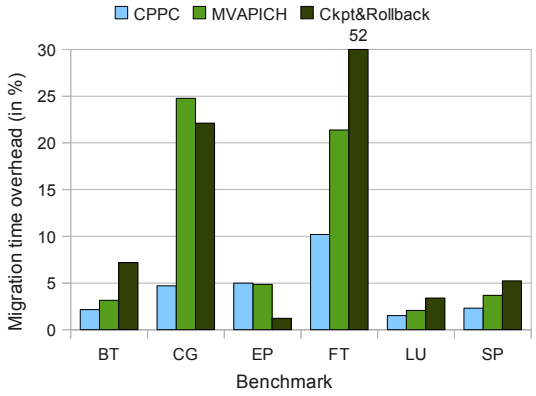


FIGURE 13. Overhead (in %) for NPB applications (class C - 16 processes) when a node is preemptively migrated (case of CPPC and MVAPICH) and when a node fails (case of Ckpt&Rollback)

when migrating only one process. In [21] a percentage of false positives smaller than 10% is reported. Thus, we can conclude that the overhead due to this issue will not be very significant.

5. RELATED WORK

Process migration may be implemented either through dynamic migration or based on the simple stop-and-restart approach [22, 23]. In this section we will focus on proposals that, like the one proposed in this work, address dynamic process migration.

Some existing approaches rely on operating system virtualization techniques. Hacker et al. [24] investigate the use of OpenVZ to perform dynamic migration of parallel applications. Chackravorty et al. [5] use Charm++ [25] and Adaptive MPI (AMPI) [26] to implement a transparent proactive fault tolerance approach. In [27] the live migration mechanism in Xen [28] is exploited to implement a live migration solution. However, the same authors reported in a later work [29] that, in HPC, solutions at the process level are more widely accepted than those based on virtualization, mainly due to the lower penalty in performance.

In [30] a process level solution through checkpointing using the previously mentioned system level checkpointing tool BLCR is presented. The proposal extends both BLCR and LAM/MPI to allow process migration. Although the authors do not present experimental results, the paper explicitly states that the checkpoint file writing has a high overhead. Wang et al. [6] reduce this overhead through a live migration solution (execution proceeds while a process image is asynchronously transferred to a spare node) at the expenses of an increase in evacuation time. The migration mechanism implemented in MVAPICH2 [20] also relies on BLCR. It takes advantage of the Remote Direct Memory Access (RDMA) in Infiniband to reduce the I/O overhead [31]. Other proposal that uses a different migration mechanism is MPI_Mitten [32], an MPI library implemented on the HPCM (High Performance Computing Mobility) middleware [33] 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.

The main contribution of our work is a proactive process migration mechanism through the extension of a checkpointing tool, the CPPC framework [8]. The main difference with previous work is that our approach is implemented at the application level, and thus it is independent of the hardware architecture, the OS, the MPI implementation used and the job submission framework. Besides, using CPPC for the implementation introduces performance advantages: 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; a consistent global state is achieved through a light and asynchronous protocol based on the safe points previously identified by the CPPC compiler, which allows some overlap between state files creation and process migration. Both features lead to a migration solution with a reduced overhead cost. Besides, unlike in system-level approaches, the new processes do not need to be completely restarted in the new nodes before the nodes that are about to fail may be freed. The terminating processes can safely finalize their execution once the reconfiguration of the communicators is finished and the checkpoint dumping is complete.

To be effective, this solution requires that failures can be anticipated accurately. However, this issue should not be seen as a limitation nowadays. Health monitoring has become a common feature in servers and HPC components. Such monitors range from processor temperature sensors to baseboard cards with a variety of sensing capabilities, including fan speeds, voltage levels and chassis temperatures. Recent studies show that, assisted by such capabilities, node failures may be predicted in large-scale systems with a high degree of accuracy [17, 21, 34–36].

6. CONCLUSIONS AND FUTURE WORK

The approach presented in this work extends CPPC to proactively migrate processes from processors when impending failures are notified, without having to restart the entire application. It has been proved to be more efficient than the classical checkpoint and rollback solution. Besides, the proposed approach makes improvements on the two most important overhead factors in other migration solutions: process coordination and I/O overhead.

A light and asynchronous protocol has been designed to achieve a global consistent state during the migration operation, avoiding, when possible, operations that lead to stalls in the processes execution.

The approach improves efficiency through the reduction of the checkpoint read/write overhead, since the use of CPPC allows for reduction in the checkpoint file sizes, and the dumping of the terminating processes state is overlapped to a certain extent with other stages of the migration operation. The experimental validation performed

has shown the efficiency and scalability of the proposal.

Another remarkable feature is that the solution is implemented at the application level, and thus it is independent of the hardware architecture, the OS or the MPI implementation used, and of any higher-level frameworks, such as job submission frameworks. Despite being implemented at the application level, the solution is transparent to the programmer, thanks to the compiler tool that automatically transforms the application source files into a fault-tolerant version with migration capabilities.

Despite the reduced checkpoint file size, write/read of the processes state continues to be the main cause of overhead of our approach. For this reason, future work will focus on the reduction of I/O cost. There exist in the literature several works that try to minimize I/O overhead. In [37] 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), to store checkpoint data [38]. SSDs offer excellent read/write throughput when compared to secondary storage and thus they can help reduce disk I/O load.

In all the experiments we have assumed that checkpoint dumping is only performed upon a migration request due to the notification of an imminent failure. In a practical scenario additional checkpoints should be included in order to cope with unpredicted failures in a traditional reactive way. The determination of optimal checkpointing intervals for reactive approaches has been studied extensively in the past [39–41]. Proactive approaches, however, improve the reliability of the application, allowing to reduce checkpoint frequency. In [39] a mathematical model is used to determine the optimal intervals in distributed systems subject to failures. Extending this model to consider the combination of reactive and proactive approaches is a promising research line.

ACKNOWLEDGEMENTS

This research was supported by the Ministry of Science and Innovation of Spain (Project TIN2010-16735) and by the Galician Government (10PXIB105180PR).

REFERENCES

[1] Iosup, A., Jan, M., Sonmez, O., and Epema, D. (2007) On the dynamic resource availability in Grids. *Proceedings of GRID 07*, Austin, TX, USA, 19–21 September, pp. 26–33. IEEE Computer Society Press, Los Alamitos.

[2] Elnozahy, E., Alvisi, L., Wang, Y.-M., and Johnson, D. (2002) A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, **34**, 375–408.

[3] Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. (2007) A job pause service under LAM/MPI+BLCR for transparent fault tolerance. *Proceedings of IPDPS 07*, Long Beach, CA, USA, 26–30 March, pp. 1–10. IEEE Computer Society Press, Los Alamitos.

[4] Salfner, F., Lenk, M., and Malek, M. (2010) A survey of online failure prediction methods. *ACM Comput. Surv.*, **42**, 10:1–10:42.

[5] Chakravorty, S., Mendes, C. L., and Kale, L. V. (2006) Proactive fault tolerance in MPI applications via task migration. *Proceedings of HiPC 06*, Bangalore, India, 18–21 December, pp. 485–496. Springer, Berlin.

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

[7] Cappello, F., Casanova, H., and Robert, Y. (2010) Checkpointing vs. migration for post-petascale supercomputers. *Proceedings of ICPP 10*, San Diego, CA, USA, 13–16 September, pp. 168–177. IEEE Computer Society Press, Los Alamitos.

[8] Rodríguez, G., Martín, M. J., González, P., Touriño, J., and Doallo, R. (2010) CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. *Concurr. Comput.-Pract. Exp.*, **22**, 749–766.

[9] Rodríguez, G., Martín, M. J., González, P., and Touriño, J. (2009) A heuristic approach for the automatic insertion of checkpoints in message-passing codes. *J. Univers. Comput. Sci.*, **15**, 2894–2911.

[10] Rodríguez, G., Martín, M. J., González, P., and Touriño, J. (2011) Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study. *Comput. J.*, **54**, 1821–1837.

[11] Chen, Y., Plank, J., and Li, K. (1997) CLIP: A checkpointing tool for message-passing parallel programs. *Proceedings of SC 97*, San Jose, CA, USA, 15–21 November, pp. 1–11. IEEE Computer Society Press, Los Alamitos.

[12] Hursey, J., Squyres, J. M., Mattox, T. I., and Lumsdaine, A. (2007) The design and implementation of checkpoint/restart process fault tolerance for Open MPI. *Proceedings of IPDPS 07*, Long Beach, CA, USA, 26–30 March, pp. 1–8. IEEE Computer Society Press, Los Alamitos.

[13] Elnozahy, E. and Plank, J. (2004) Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secur. Comput.*, **1**, 97–108.

[14] Cores, I., Rodríguez, G., Martín, M. J., and González, P. (2012) Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. *Proceedings of ISPA 12*, Madrid, Spain, 10–13 July, pp. 371–378. IEEE Computer Society Press, Los Alamitos.

[15] Cores, I., Rodríguez, G., González, P., and Martín, M. J. (2011) An application level approach for proactive process migration in MPI applications. *Proceedings of PDCAT 11*, Gwangju, Korea, 20–22 October, pp. 400–405. IEEE Computer Society Press, Los Alamitos.

[16] National Aeronautics and Space Administration. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. Last accessed July 2012.

[17] Tan, Y., Gu, X., and Wang, H. (2010) Adaptive system anomaly prediction for large-scale hosting infrastructures. *Proceedings of PODC 10*, Zurich, Switzerland, 25–28 July, pp. 173–182. ACM, New York.

[18] Laurence Berkeley National Laboratory. Berkeley Lab Checkpoint/Restart. `https://ftg.lbl.gov/CheckpointRestart/`. Last accessed July 2012.

[19] Gupta, R., Beckman, P., Park, B. H., Lusk, E., Hargrove, P., Geist, A., Lumsdaine, A., and Dongarra, J. (2009) CIFTS:

A coordinated infrastructure for fault-tolerant systems. *Proceedings of ICPP 09*, Vienna, Austria, 22–25 September, pp. 237–245. IEEE Computer Society Press, Los Alamitos.

[20] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu`. Last accessed July 2012.

[21] Gu, J., Zheng, Z., Lan, Z., White, J., Hocks, E., and Park, B.-H. (2008) Dynamic meta-learning for failure prediction in large-scale systems: A case study. *Proceedings of ICPP 08*, Portland, OR, USA, 8–12 September, pp. 157–164. IEEE Computer Society Press, Los Alamitos.

[22] UW-CS-TR-1346 (1997). Checkpoint and migration of UNIX processes in the Condor distributed processing system. University of Wisconsin-Madison, Madison, WI, USA.

[23] Cao, J., Li, Y., and Guo, M. (2005) Process migration for MPI applications based on coordinated checkpoint. *Proceedings of ICPADS 05*, Fukuoka, Japan, 20–22 July, pp. 306–312. IEEE Computer Society Press, Los Alamitos.

[24] Hacker, T. J., Romero, F., and Nielsen, J. J. (2012) Secure live migration of parallel applications using container-based virtual machines. *Int. J. Space-Based and Situated Comput.*, **2**, 45–57.

[25] Kale, L. V. and Krishnan, S. (1996) Charm++: Parallel programming with message-driven objects. In Wilson, G. V. and Lu, P. (eds.), *Parallel Programming using C++*, pp. 175–213. MIT Press, Cambridge, MA, USA.

[26] Huang, C., Lawlor, O., and Kalé, L. V. (2003) Adaptive MPI. *Proceedings of LCPC 03*, College Station, TX, USA, 2–4 October, pp. 306–322. Springer, Berlin.

[27] Nagarajan, A. B., Mueller, F., Engelmann, C., and Scott, S. L. (2007) Proactive fault tolerance for HPC with Xen virtualization. *Proceedings of ICS 07*, Seattle, WA, USA, 17–21 June, pp. 23–32. ACM, New York.

[28] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003) Xen and the art of virtualization. *Proceedings of SOSP 03*, Bolton, NY, USA, 19–22 October, pp. 164–177. ACM, New York.

[29] Engelmann, C., Vallee, G. R., Naughton, T., and Scott, S. L. (2009) Proactive fault tolerance using preemptive migration. *Proceedings of PDP 09*, Weimar, Germany, 18–20 February, pp. 252–257. IEEE Computer Society Press, Los Alamitos.

[30] Singh, R. and Graham, P. (2008) Performance driven partial checkpoint/migrate for LAM-MPI. *Proceedings of HPCS 08*, Québec City, Canada, 9–11 June, pp. 110–116. IEEE Computer Society Press, Los Alamitos.

[31] Ouyang, X., Marcarelli, S., Rajachandrasekar, R., and Panda, D. K. (2010) RDMA-Based job migration framework for MPI over InfiniBand. *Proceedings of CLUSTER 10*, Heraklion, Greece, 20–24 September, pp. 116–125. IEEE Computer Society Press, Los Alamitos.

[32] Du, C. and Sun, X.-H. (2006) MPI-Mitten: Enabling migration technology in MPI. *Proceedings of CCGRID 06*, Singapore, 16–19 May, pp. 11–18. IEEE Computer Society Press, Los Alamitos.

[33] Du, C., Sun, X.-H., and Chanchio, K. (2003) HPCM: A pre-compiler aided middleware for the mobility of legacy code. *Proceedings of CLUSTER 03*, Hong Kong, China, 1–4 December, pp. 180–187. IEEE Computer Society Press, Los Alamitos.

[34] Liang, Y., Zhang, Y., Xiong, H., and Sahoo, R. (2007) Failure prediction in IBM BlueGene/L event logs. *Proceedings of ICDM 07*, Omaha, NE, USA, 28–31 October, pp. 583–588. IEEE Computer Society Press, Los Alamitos.

[35] Sahoo, R. K., Oliner, A. J., Rish, I., Gupta, M., Moreira, J. E., Ma, S., Vilalta, R., and Sivasubramaniam, A. (2003) Critical event prediction for proactive management in large-scale computer clusters. *Proceedings of KDD 03*, Washington, DC, USA, 24–27 August, pp. 426–435. ACM, New York.

[36] Hacker, T. J., Romero, F., and Carothers, C. D. (2009) An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput.*, **69**, 652–665.

[37] Ouyang, X., Rajachandrasekar, R., Besseron, X., and Panda, D. K. (2011) High performance pipelined process migration with RDMA. *Proceedings of CCGRID 11*, Newport Beach, CA, USA, 23–26 May, pp. 314–323. IEEE Computer Society Press, Los Alamitos.

[38] Li, M., Vazhkudai, S. S., Butt, A. R., Meng, F., Ma, X., Kim, Y., Engelmann, C., and Shipman, G. M. (2010) Functional partitioning to optimize end-to-end performance on many-core architectures. *Proceedings of SC 10*, New Orleans, LA, USA, 13–19 November, pp. 1–12. IEEE Computer Society Press, Los Alamitos.

[39] Gelenbe, E., Finkel, D., and Tripathi, S. K. (1986) Availability of a distributed computer system with failures. *Acta Inform.*, **23**, 643–655.

[40] Gelenbe, E. and Hernández, M. (1989) Optimum checkpoints with age dependent failures. *Acta Inform.*, **27**, 519–531.

[41] Vaidya, N. (1997) Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, **46**, 942–947.