An Application-Level Solution for the Dynamic Reconfiguration of MPI Applications

Iván Cores[1], Patricia González[1], Emmanuel Jeannot[2], María J. Martín[1(✉)], and Gabriel Rodríguez[1]

[1] Grupo de Arquitectura de Computadores, Universidade da Coruña, A Coruña, Spain
mariam@udc.es
[2] INRIA Bordeaux Sud-Ouest, Bordeaux, France

Abstract. Current parallel environments aggregate large numbers of computational resources with a high rate of change in their availability and load conditions. In order to obtain the best performance in this type of infrastructures, parallel applications must be able to adapt to these changing conditions. This paper presents an application-level proposal to automatically and transparently adapt MPI applications to the available resources. The architecture includes: automatic code transformation of the parallel applications, a system to reschedule processes on available nodes, and migration capabilities based on checkpoint-and-restart techniques to move selected processes to target nodes. Experimental results show a good degree of adaptability and a good performance in different availability scenarios.

Keywords: HPC · MPI · Checkpointing · Migration · Scheduling

1 Introduction

The resources availability of large-scale distributed systems may vary during a job execution, making malleable applications, that is, parallel programs that are able to adapt their execution to the number of available processors at runtime, specially appealing. Malleable jobs provide important advantages for the final users and the whole system, like a higher productivity and a better response time [3,9], or a greater resilience to node failures [5]. These characteristics will allow to improve the use of resources, which will have a direct effect on the energy consumption required for the execution of applications, resulting in both cost savings and a greener computing.

High performance computing (HPC) is nowadays dominated by the MPI paradigm. Most MPI applications follow the SPMD (single program, multiple data) programming model and they are executed in HPC systems by specifying a fixed number of processes running on a fixed number of processors. The resource allocation is statically specified during job submission, and maintained constant during the entire execution. Thus, MPI applications are unable to dynamically adapt to changes in resource availability.

I. Dutra et al. (Eds.): VECPAR 2016, LNCS 10150, pp. 191–205, 2017.
DOI: 10.1007/978-3-319-61982-8_18

The aim of this work is to propose a solution to transform existing MPI applications into malleable jobs, that is, jobs that are capable of adapting their executions to changes in the environment. The proposed solution is based on process migration, thus, if a node becomes unavailable, the processes on that node will be migrated to other available ones, overloading nodes when necessary. Checkpoint and restart is used to implement this migration. The state of each process to be migrated is stored into memory and transferred to a new available node. Afterwards, this state is recovered by a newly created process, which continues the execution. To this end, the proposal is implemented at the application-level, extending the functionalities of the CPPC (ComPiler for Portable Checkpointing) framework [18]. CPPC is an application-level checkpointing tool, available under GNU general public license (GPL) at http://cppc.des.udc.es, that appears to the final user as a compiler tool and a runtime library. At compile time, the CPPC source-to-source compiler automatically transforms a parallel code into an equivalent version with calls to the CPPC library to instrument the dynamic reconfiguration.

The structure of this paper is as follows. Section 2 describes related work. Section 3 describes the solution proposed to automatically and transparently transform existing MPI application into malleable jobs. Section 4 evaluates the performance of the proposal. Finally, Sect. 5 concludes the paper.

2 Related Work

There are several proposals in the literature that use a stop-and-restart approach to implement malleable MPI applications [1,16,20]. However, stop and restart solutions imply a job requeueing, with the consequent loss of time. Dynamic reconfiguration [8,12,17,22], on the other hand, allows to change the number of processors while the program is running without having to stop and relaunch the application. Most of these solutions [12,17,22] change the number of processes to adapt to the number of available processors, which implies a redistribution of the data and, thus, they are very restrictive with the kind of application they support. On the other hand, in AMPI [8] the number of processes is preserved and the application adapts to changes in the number of resources through migration based on virtualization.

Besides AMPI, there exist in the literature different research works that provide process migration through the use of virtualization technologies [7,13]. However, virtualization solutions present important performance penalties due to larger memory footprints [21]. Moreover, the performance of MPI applications relies heavily on the performance of communications. Currently virtualization of CPU and memory resources presents a low overhead. However, the efficient virtualization of network I/O is still work in progress. For instance, recent results in migration over Infiniband networks [6] show very high overhead, strong scalability limitations and tedious configuration issues.

3 Reconfiguration of MPI Applications

The aim of this work is to build MPI applications that are able to dynamically adapt their execution to changes in the resource availability. The following subsections describe the main components of the proposed solution: (1) the triggering of the reconfiguration operation; (2) the scheduling algorithm implemented to allow the application to autonomously decide which processes should be moved and to which target nodes; and (3) the migration operation itself. The main steps of the reconfiguration process are depicted in Fig. 1.

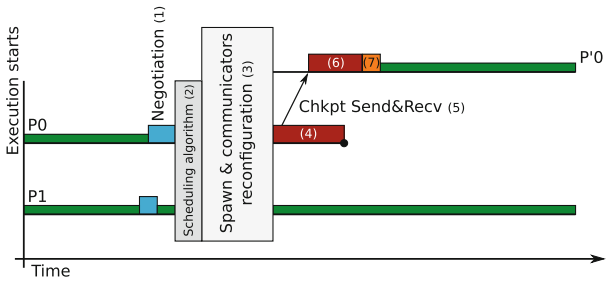

Fig. 1. Steps in the reconfiguration operation: (1) Negotiation to select a single safe location to trigger the reconfiguration; (2) Scheduling algorithm to decide the processes to be migrated and the new allocations; (3) Spawning of new processes and reconfiguration of the communicators; (4) Checkpointing of the migrating processes; (5) Sending of the checkpoint files; (6) Recovering the state from the checkpoint files; and (7) Effective recovering of the application state. Steps (4) to (6) are partially overlapped.

3.1 Triggering the Reconfiguration Operation

The proposed solution is based on dynamic live process migration. If a node becomes unavailable, the processes on that node will be migrated to other available ones (without stopping the application), overloading nodes when necessary. Our proposal relies on a monitoring system that provides dynamical information about the available resources. There are in the literature many proposals for different environments and objectives. For this work we assume that an availability file is set up for each malleable MPI job. This file contains the names of all the nodes that are assigned to execute the MPI job together with their number of available cores. A change in this file activates a flag in the CPPC controller to start the reconfiguration of the execution. During this reconfiguration, some MPI processes will be migrated and, thus, communication groups must be rebuilt. 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 restrict the points

at which the migration can be started, making the reconstruction of the communicators, and thus the reconfiguration, 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. However, conducting the reconfiguration from different safe points in different processes may lead to inconsistencies. Therefore, a negotiation protocol is needed at runtime to select a single safe location as the place to trigger the reconfiguration operation to achieve a consistent global state after migration. This location will be the next safe point to be reached by the process that has advanced the farthest in the execution. Each process communicates to all other processes the last safe point it has crossed. One-sided asynchronous MPI communications are used so that processes may continue running without synchronizations during the negotiation, overlapping negotiation operations with execution progress and avoiding deadlocks.

3.2 Scheduling Algorithm

Previous to the start of the migration operation, the processes to be migrated and their mapping to the available resources need to be selected.

In MPI applications the communication overhead plays an important role in the global performance of the parallel application. To be able to migrate the processes efficiently we need to know the *affinity* between the processes so as to map those with a high communication rate as close to each other as possible. For this aim TreeMatch [11] and Hwloc [2] are used. TreeMatch is an algorithm that obtains the optimized process placement based on the communication pattern among the processes and the hardware topology of the underlying computing system. It tries to minimize the communications at all levels, including network, memory and cache hierarchy. It takes as input both a matrix modeling the communications among the processes, and a representation of the topology of the system. The topological information, represented as a tree, is provided by Hwloc and obtained dynamically during application execution. TreeMatch returns as output an array with the core ID that should be assigned to each process.

An example of the output of TreeMatch is given in Fig. 2. On the left, a communication matrix representing the affinity between processes is given as input: the darker the dot the higher the communication volume and hence the affinity. TreeMatch computes the permutation (σ) of the processes such that the cores with high affinity are mapped close together on the tree representing the target topology. After applying the permutation the communication matrix on the right is obtained.

The communication matrix needed by TreeMatch is obtained dynamically, just before the scheduling algorithm is triggered, using a monitoring component developed for Open MPI and integrated in an MCA (Modular Component Architecture) framework called *pml* (point-to-point management layer). This component, if activated at launch time (through the `mpiexec` option `--mca pml_monitoring_enable`), monitors all the communications at the lower level of Open MPI (i.e., once collective communications have been decomposed into

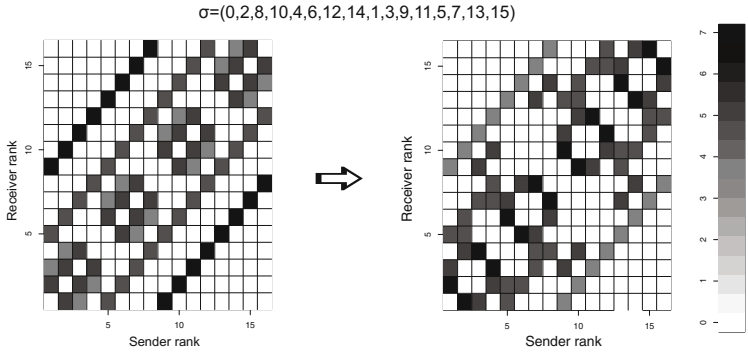

Fig. 2. TreeMatch example for a binary tree of 4 levels and 16 leaves (cores).

send/recv communications). Therefore, contrary to the MPI standard profiling interface (`PMPI`) approach where the MPI calls are intercepted, here the actual point-to-point communications that are issued by Open MPI are monitored, which is much more precise. This monitoring component was previously developed by one of the authors and it will be released in Open MPI 2.0.

To evaluate the overhead of this monitoring, the execution of the LU NAS benchmark with and without monitoring have been compared. The LU kernel has been selected since it is the one that sends the largest number of messages. Results for this kernel are shown in Table 1. Shown times are the average of 10 runs. Results using 16 and 64 processes using classes A, B and C show that the overhead is very low (less than 0.7%). Moreover, it decreases with the class of the kernel (i.e., the problem size), being the overhead to manage one message 1 µs or less.

TreeMatch focuses on minimizing the communication cost in a parallel execution. Thus, if TreeMatch is directly applied to find the processes mapping during a reconfiguration phase, it could lead to a complete replacement of all the application processes. This would involve unnecessary process migrations and, thus,

Table 1. Monitoring overhead for the LU NAS Kernel on nodes with 2 quad-core Nehalem Intel Xeon processors interconnected by an InfiniBand QDR network.

Class	Number of processes	Total number of messages	Number of mess. per processes	Exec. time (s)	Monitoring exec. time (s)	Overhead
A	16	380630	23789.375	5.696	5.72	0.42%
B	16	609542	38096.375	23.155	23.189	0.15%
C	16	970982	60686.375	90.665	90.727	0.10%
A	64	1777226	27769.156	1.732	1.744	0.69%
B	64	2845482	44460.656	6.522	6.567	0.69%
C	64	4532202	70815.656	25.335	25.374	0.15%

unnecessary overheads. To avoid this behavior, a two-step mapping algorithm was designed. The first step decides the number and the specific processes to be migrated. The second step finds the best target nodes and cores to place these processes. An interesting feature of TreeMatch is that the topology given as an input can be a real machine topology or a virtual topology designed to separate groups of processes in clusters such that communications within clusters are maximized while communications outside the clusters are minimized.

- **Step 1: identify processes to migrate**. A process should be migrated either because it is running on nodes that are going to become unavailable, or because it is running on oversubscribed nodes and new resources have become available. To know the number of processes that need to be migrated, all processes exchange, via MPI communications, the numbers of the node and core in which they are currently running. Then, using this information, each application process calculates the current computational load of each node listed in the availability file associated to the application. A *load* array is computed, where $load(i)$ is the number of processes that are being executed in node n_i. Besides, each process also calculates the maximum number of processes that could be allocated to each node n_i in the new configuration:

$$maxProcs(i) = \left\lceil nCores(i) \times \frac{N}{nTotalCores} \right\rceil$$

  where $nCores(i)$ is the number of available cores of node n_i, N is the number of processes of the MPI application, and $nTotalCores$ is the number of total available cores. If $load(i) > maxProcs(i)$ then $load(i) - maxProcs(i)$ processes have to be migrated. If the node is no longer available, $maxProcs(i)$ will be equal to zero and all the processes running in that node will be identified as migrating processes. Otherwise, TreeMatch is used to identify the migrating processes. The aim is to maintain in each node the most related processes according to the application communication pattern. Figure 3 illustrates an example with two 16-core nodes executing a 56-process application in an oversubscribed scenario. When two new nodes become available, 12 processes per node should be migrated to the new resources. To find the migrating processes, TreeMatch is queried once for each oversubscribed node. The input is a virtual topology breaking down the node into two virtual ones: one with maxProcs(i) cores and the other with load(i) − maxProcs(i) cores. TreeMatch uses in runtime this virtual topology, and a sub-matrix with the communication pattern between the processes currently running on the node, to identify the processes to be migrated, that is, those mapped to the second virtual node.
- **Step 2: identify target nodes**. Once the processes to be migrated are identified, the target nodes (and the target cores inside the target nodes) to place these processes have to be found. TreeMatch is again used to find the best placement for each migrating process. It uses a sub-matrix with the communication pattern of the migrating processes, and a virtual topology built from the real topology of the system but restricted to use only the

potential target nodes in the cluster. The potential targets are those nodes that satisfy $load(i) < maxProcs(i)$. They can be empty nodes, nodes already in use but with free cores, or nodes that need to be oversubscribed. Since TreeMatch only allows the mapping of one process per core, if there are no sufficient real target cores to allocate the migrating processes, a virtual topology will simulate $maxProcs(i) - load(i)$ extra cores in the nodes that need to be oversubscribed. Figure 4 illustrates the second step of the algorithm for the same example of Fig. 3. In this example, the virtual topology used consists of the new available nodes in the system, two 16-core nodes to map the 24 processes. After executing TreeMatch, the target cores and, therefore, the target nodes for the migrating processes obtained in step 1 are identified and CPPC can be used to perform the migration.

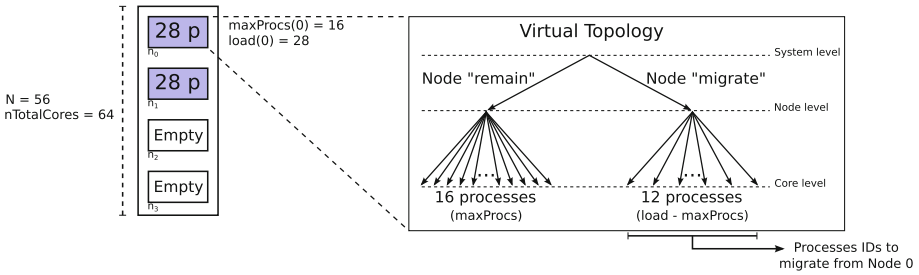

Fig. 3. Step 1: identifying processes to be migrated. Virtual topology built to migrate 12 processes from a 16-core node where 28 processes are running (16 processes remain and 12 processes migrate).

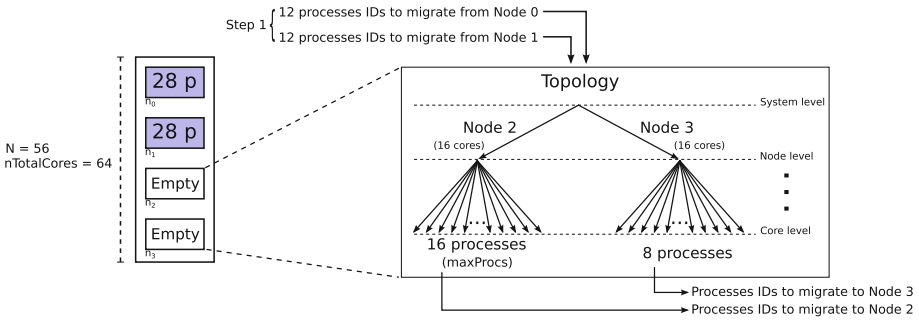

Fig. 4. Step 2: identifying target nodes. Topology built to map the migrating processes selected in step 1 to the empty cores in the system.

3.3 Migration Operation

Once the mapping of migrated processes to available resources is decided, the migration operation can start. First, 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. Then, the migrating processes save their state, storing it into memory. 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 restart the execution by reading the checkpoint files and recovering the application state. This is achieved by delegating to CPPC and employing its native capabilities. The procedure is depicted in Fig. 1.

Initially the new spawned processes are not bound to any specific core. The TreeMatch assignment is sent to the new processes together with the checkpoint file and CPPC performs the binding via the Hwloc library.

To minimize the overhead associated to the I/O operations needed for the migration, the checkpoint files are split into several chunks and transferred in a pipelined fashion, overlapping the writing in the terminating processes with the reading in the newly spawned ones [19].

4 Experimental Results

This section aims to show the feasibility of the proposal and to evaluate the cost of the reconfiguration whenever a change in the resource availability occurs. A multicore cluster was used to carry out these experiments. It consists of 16 nodes, each powered by two octa-core Intel Xeon E5-2660 CPUs with 64 GB of RAM. The cluster nodes are connected through an InfiniBand FDR network. The working directory is mounted via network file system (NFS) and it is connected to the cluster by a Gigabit Ethernet network.

The application testbed is composed of six out of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [14] (NPB from now on). The IS and EP benchmarks were discarded due to their low execution times. For all the executions the benchmark size used was class C. The Himeno benchmark [10] was also tested. Himeno uses the Jacobi iteration method to solve the Poissons's equation, evaluating the performance of incompressible fluid analysis code, being a benchmark closer to real applications.

The MPI implementation used was Open MPI v1.8.1 [15] modified to enable dynamic monitoring. The `mpirun` environment has been tuned using MCA parameters to allow the reconfiguration of the MPI jobs. Specifically, the parameter `orte_allowed_exit_without_sync` has been set to allow some processes to continue their execution when other processes have finished their execution safely. The parameter `mpi_yield_when_idle` was also set to force degraded execution mode and, thus, to allow progress in an oversubscribed scenario.

To evaluate the feasibility of the proposed solution and its performance, different scenarios have been forced during the execution of the applications. Figure 5 illustrates these scenarios. The applications were initially launched in

a 64-process configuration running on 4 available nodes of the cluster (16 cores per node). Then, after a time, one of the nodes becomes unavailable. In this scenario, the 16 processes running on the first node should be moved to the empty node, and the application execution continues in a 4 node configuration. After a while, the 4 nodes where the application is running start to become unavailable sequentially, first one, then another, without spare available nodes to replace them, until only one node is available and the 64 processes are running on it. Finally, in a single step, the last node fails but 4 nodes become available again, and the processes are migrated to return again to using 4 nodes. To demonstrate the feasibility of the solution, the iteration time was measured across the execution in those scenarios. Measuring iteration time allows to have a global vision of the instantaneous performance impact.

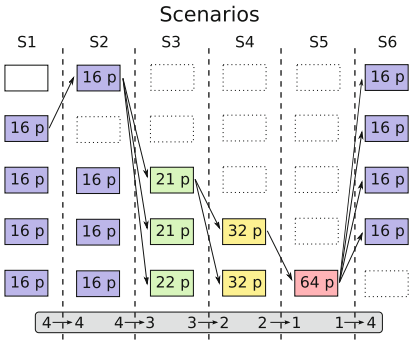

Fig. 5. Selected scenarios to show the feasibility of the proposal and evaluate the migration and scheduling cost.

Figure 6 shows the results for all the benchmarks in the scenarios illustrated in Fig. 5. These results demonstrate that, using the proposed solution, the applications are capable of adjusting their execution to changes in the environment. The high peaks in these figures correspond to reconfiguration points. For comparison purposes, the compute times that would be attained if the application could adjust its granularity to the available resources[1], instead of oversubscribing them without modifying the original number of processes are also shown (green line). The overhead that would introduce the data distribution needed to adjust the application granularity is not shown in the figure.

Table 2 details the main impacting steps in the reconfiguration overhead for all the NPB applications. As shown in Fig. 1, the iteration time when a reconfiguration is performed can be broken down into the following stages:

– *Negotiation*: execution time of the negotiation protocol used to reach consensus on the reconfiguration point.

[1] This time is measured executing the application with a different number of processes depending on the hardware available (16 processes version when only 1 node is available, 32 processes version when 2 nodes are available, etc.).

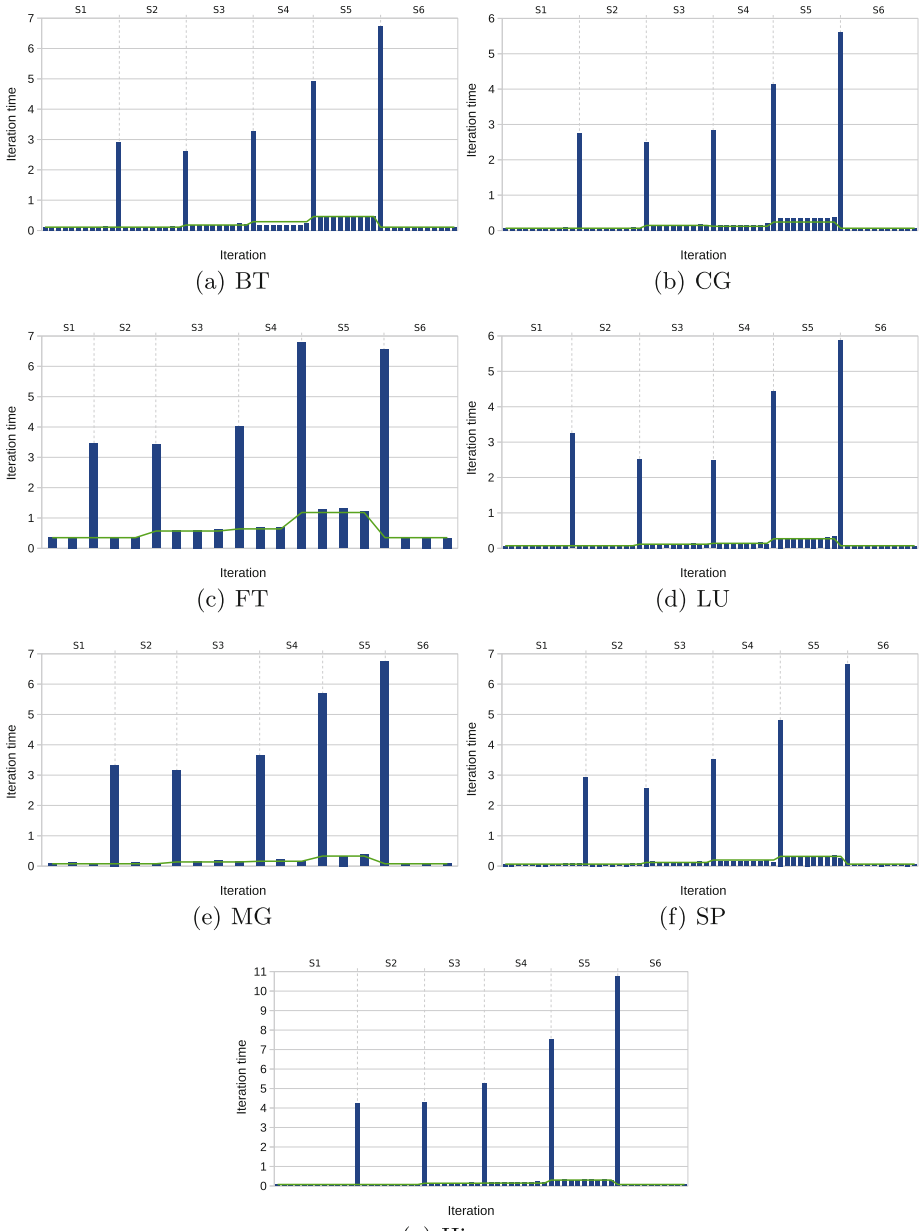

Fig. 6. Iteration execution times in the scenarios illustrated in Fig. 5. (Color figure online)

- *Scheduling*: execution time of the scheduling algorithm to identify processes to be moved and target nodes.
- *Spawn&Rec*: execution time of the spawn function and the reconfiguration of the communicators.
- *ChkptTransfer&Read*: average time to write the checkpoint files in the terminating processes, transfer them to target nodes, and read them in newly spawned processes.
- *Restart*: average time to complete the restart of the application once the checkpoint files have been read.
- *Compute*: the computational time of the iteration where the reconfiguration takes place.

Table 2. Execution time (s) of the reconfiguration phases.

	scenarios	NPB applications						*Himeno*
		BT	*CG*	*FT*	*LU*	*MG*	*SP*	
Negotiation	$1 \rightarrow 2$	0.88	1.18	0.01	1.08	0.93	0.94	1.12
	$2 \rightarrow 3$	0.80	0.85	0.01	0.86	0.94	0.81	0.90
	$3 \rightarrow 4$	0.88	0.89	0.14	0.53	0.94	0.86	0.95
	$4 \rightarrow 5$	1.06	1.11	0.02	1.15	1.07	1.04	1.09
	$5 \rightarrow 6$	1.86	1.94	0.02	1.90	1.88	1.87	1.95
Spawn&Rec	$1 \rightarrow 2$	1.32	1.18	1.44	1.61	1.28	1.29	1.13
	$2 \rightarrow 3$	1.15	1.18	1.11	1.08	1.09	1.14	0.99
	$3 \rightarrow 4$	1.43	1.43	1.32	1.31	1.35	1.59	1.48
	$4 \rightarrow 5$	2.23	2.14	2.14	2.17	2.38	2.07	2.12
	$5 \rightarrow 6$	3.27	3.19	3.25	3.28	3.20	3.22	3.23
ChkptTransfer&Read	$1 \rightarrow 2$	0.35	0.10	0.39	0.16	0.44	0.39	1.44
	$2 \rightarrow 3$	0.37	0.11	0.43	0.18	0.42	0.40	1.59
	$3 \rightarrow 4$	0.48	0.14	0.47	0.21	0.54	0.54	1.96
	$4 \rightarrow 5$	0.75	0.23	0.84	0.36	0.84	0.93	2.92
	$5 \rightarrow 6$	1.01	0.28	1.41	0.47	1.41	1.06	4.96
Restart	$1 \rightarrow 2$	0.15	0.01	0.04	0.01	0.04	0.12	0.25
	$2 \rightarrow 3$	0.05	0.01	0.20	0.02	0.03	0.08	0.46
	$3 \rightarrow 4$	0.24	0.01	0.26	0.02	0.03	0.30	0.47
	$4 \rightarrow 5$	0.36	0.02	0.41	0.02	0.06	0.30	0.57
	$5 \rightarrow 6$	0.36	0.01	0.41	0.01	0.03	0.29	0.45

The *Negotiation* phase depends on the application as in this phase MPI one-sided communications are used and the progress of these remote operations is affected by the MPI calls inside the application. These times could be lower using other MPI implementations and/or computer architectures [4].

The largest contribution to the reconfiguration cost is due to the *Spawn&Rec* step. The time spent in the spawn function depends on the number of spawned processes and the degree of oversubscription. The more processes to be migrated, the larger the overhead of this phase. This can be observed comparing the overhead associated to the reconfiguration from scenario 1 to scenario 2, where 16 processes are moved to an empty target node, and the overhead associated to the reconfiguration from scenario 5 to scenario 6, where 64 processes are moved to 4 empty target nodes. When target nodes are oversubscribed, the computation time of each process is penalized and so is the *Spawn&Rec* phase, specially affected due to their collective communications. This can be observed in the increase that the overhead of the *Spawn&Rec* phase suffers in the reconfiguration from scenario 2 to scenario 3, from scenario 3 to scenario 4, from scenario 4 to scenario 5, and from scenario 5 to scenario 6, where 16, 21, 32 and 64 processes are migrated each time, oversubscribing the surviving nodes. Finally, since this phase involves different collective communications, its time depends on the total number of processes in the application. This can be observed in Table 3, that shows the overhead of the *Spawn&Rec* step when migrating 16 processes to an empty target node with a different number of processes in the application.

Table 3. Overhead (in seconds) of the *Spawn&Rec* step when spawning 16 new processes vs total number of processes in the application.

NPB	Number of total processes			
	16	32/36	64	128/121
BT	0.97	0.99	1.32	1.57
CG	0.98	1.01	1.18	1.79
FT	0.96	1.07	1.44	1.75
LU	1.00	1.01	1.61	1.89
MG	1.02	1.01	1.28	1.63
SP	0.99	1.00	1.29	1.72
Himeno	0.99	1.01	1.13	1.96

The *ChkptTransfer&Read* step also impacts significantly the reconfiguration overhead. The I/O operations are recognized to be one of the main impacting factors in the performance of migration operations, specially in those associated to checkpoint solutions. 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. Table 4 shows the checkpoint sizes per process and the total data size transferred between nodes when migrating 16, 32 and 64 processes. The total amount of data varies between 127 MB for CG migrating a single node (16 processes) and 10.42 GB for Himeno when migrating 4 nodes (64 processes). By means of a pipelined approach [19] that overlaps the state file writing in the terminating processes, the data transfer through the network,

Table 4. Transfer size (checkpoint size in MB).

NPB	Checkpoint size per process	Total data size migrated		
		16 proc.	32 proc.	64 proc.
BT	33.15	530.45	1060.90	2121.80
CG	7.93	126.97	253.95	507.90
FT	48.09	769.50	1539.01	3078.02
LU	15.48	247.74	495.48	918.96
MG	39.26	628.19	1256.39	2512.78
SP	32.12	513.99	1027.99	2055.98
Himeno	166.71	2667.36	5334.72	10669.44

and the state file read in the new processes, the proposed solution achieves to significantly reduce this impact.

The *Restart* step is a small contributor to the reconfiguration overhead. An important part of this time is derived from the negotiation protocol used. During the negotiation phase each process specifies a memory region (window) that it exposes to others. Since the MPI communicators of the application have been reconfigured, at restart time the old MPI windows have to be closed and new ones have to be created. Although not as impacting as the spawning function, the overhead of this operation is not negligible.

Finally, the *Scheduling* phase is negligible for all the NPB applications, being always smaller than 0.1 s. Thus, these times are not included in the table.

5 Concluding Remarks

In this paper a proposal to automatically and transparently adapt MPI applications to available resources is proposed. The solution relies on a previous application-level migration approach, incorporating a new scheduling algorithm based on TreeMatch, Hwloc and dynamic communication monitoring, to obtain well balanced nodes while preserving performance as much as possible.

The experimental evaluation of the proposal shows successful and efficient operation, with an overhead of a few seconds during reconfiguration, which will be negligible in large applications with a more realistic reconfiguration frequency.

Proposals like the one in this paper will be of particular interest in future large scale computing systems, since applications that are able to dynamically reconfigure themselves to adapt to different resource scenarios will be key to achieve a tradeoff between energy consumption and performance.

Acknowledgments. This research was partially supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P), by the Galician Government and FEDER funds of the EU (consolidation program of competitive reference groups GRC2013/055) and by the EU under the COST programme Action IC1305, Network for Sustainable Ultrascale Computing.

References

1. Agbaria, A., Friedman, R.: Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. Cluster Comput. **6**(3), 227–236 (2003)
2. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 180–186 (2010)
3. Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., Epema, D.: Scheduling malleable applications in multicluster systems. In: 2007 International Conference on Cluster Computing (CLUSTER), pp. 372–381 (2007)
4. Cores, I., Rodríguez, G., Martín, M.J., González, P.: Achieving checkpointing global consistency through a hybrid compile time and runtime protocol. Procedia Comput. Sci. **18**, 169–178 (2013). International Conference on Computational Science (ICCS)
5. George, C., Vadhiyar, S.S.: ADFT: an adaptive framework for fault tolerance on large scale systems using application malleability. Procedia Comput. Sci. **9**, 166–175 (2012). International Conference on Computational Science (ICCS)
6. Guay, W.L., Reinemo, S.A., Johnsen, B.D., Yen, C.H., Skeie, T., Lysne, O., Tørudbakken, O.: Early experiences with live migration of SR-IOV enabled infiniband. J. Parallel Distrib. Comput. **78**, 39–52 (2015)
7. Hacker, T.J., Romero, F., Nielsen, J.J.: Secure live migration of parallel applications using container-based virtual machines. Int. J. Space Based Situated Comput. **2**(1), 45–57 (2012)
8. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24644-2_20
9. Hungershofer, J.: On the combined scheduling of malleable and rigid jobs. In: Computer Architecture and High Performance Computing (SBAC-PAD), pp. 206–213 (2004)
10. Information Technology Center, RIKEN. HIMENO Benchmark. http://accc.riken.jp/2444.htm. Accessed Aug 2016
11. Jeannot, E., Mercier, G.: Near-optimal placement of MPI processes on hierarchical NUMA architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 199–210. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15291-7_20
12. Martín, G., Singh, D.E., Marinescu, M.C., Carretero, J.: Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. Parallel Comput. **46**, 60–77 (2015)
13. Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive fault tolerance for HPC with Xen virtualization. In: International Conference on Supercomputing (ICS), pp. 23–32 (2007)
14. National Aeronautics and Space Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. Accessed Aug 2016
15. Open MPI Team. Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/. Accessed Aug 2016
16. Raveendran, A., Bicer, T., Agrawal, G.: A framework for elastic execution of existing MPI programs. In: IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW), pp. 940–947 (2011)

17. Ribeiro, F.S., Nascimento, A.P., Boeres, C., Rebello, V.E.F., Sena, A.C.: Autonomic malleability in iterative MPI applications. In: Computer Architecture and High Performance Computing (SBAC-PAD), pp. 192–199 (2013)
18. Rodríguez, G., Martín, M.J., González, P., Touriño, J., Doallo, R.: CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. Concurr. Comput. Pract. Exper. **22**(6), 749–766 (2010)
19. Rodríguez, M., Cores, I., González, P., Martín, M.J.: Improving an MPI application-level migration approach through checkpoint file splitting. In: Computer Architecture and High Performance Computing (SBAC-PAD), pp. 33–40 (2014)
20. Vadhiyar, S.S., Dongarra, J.J.: SRS - a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process. Lett. **13**(02), 291–312 (2003)
21. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration and back migration in HPC environments. J. Parallel Distrib. Comput. **72**(2), 254–267 (2012)
22. Weatherly, D.B., Lowenthal, D.K., Nakazawa, M., Lowenthal, F.: Dyn-MPI: supporting MPI on non dedicated clusters. In: ACM/IEEE Conference on High Performance Networking and Computing (SC), p. 5 (2003)