An application-level solution for the dynamic reconfiguration of MPI applications

Iván Cores, Patricia Gonzalez, Emmanuel Jeannot, Maria J Martin, Gabriel Rodriguez

An application-level solution for the dynamic reconfiguration of MPI applications

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

[1] Grupo de Arquitectura de Computadores
Universidade da Coruña, Spain
[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. Experimental results show a good degree of adaptability and a good performance in different availability scenarios.

Keywords: HPC, MPI, Checkpointing, Migration, Scheduling

1 Introduction

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 over 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.

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. There are several proposals in the literature that use a stop-and-restart approach to implement malleable MPI applications [12, 9, 1]. However, stop and restart solutions imply a job requeueing, with the consequent loss of time. Dynamic reconfiguration [7, 14, 10, 4], 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 [7, 14, 10] change the number or 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 [4] 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. However, virtualization solutions present important performance penalties due to larger memory footprints [13]. 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 [3] show very high overhead, strong scalability limitations and tedious configuration issues.

In this paper we propose a dynamic reconfiguration solution based on process migration of non-virtualized MPI processes. In [11] an application-level checkpointing tool, CPPC (ComPiler for Portable Checkpointing), was extended to achieve efficient dynamic migration of MPI processes. In this work, we present a mapping algorithm that decides which processes should be migrated and where they should be allocated based on their affinity (gathered through dynamic monitoring) and the underlying hardware topology. The proposed approach is applicable to general SPMD MPI applications. It does not need user or system interaction and it allows both to shrink or to expand the number of processors.

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

2 Reconfiguration of MPI applications

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. 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 [6] 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 is provided by Hwloc (represented as a tree) and it is obtained dynamically during application execution. TreeMatch returns as output an array with the core ID that should be assigned to each process.

2.1 Dynamic communication monitoring

The communication matrix needed by TreeMatch is obtained dynamically, just before the scheduling algorithm is triggered, using a monitoring component developed for Open MPI that monitors all the communications at the lower level of Open MPI (i.e., once collective communications have been decomposed into 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.

The overhead of this monitoring is very low, less than 0.7% on the LU[3]NAS benchmark [8].

2.2 Mapping processes to cores using TreeMatch

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, 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 group of processes in clusters such that communication within clusters are maximized while communication outside the clusters are minimized.

- **Step1: identify processes to migrate**. A process should be migrated either because it is running in nodes that are going to become unavailable, or because it is running in 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:

[3] The LU kernel is the one that sends the largest number of messages from among all the 8 MPI NAS benchmarks.

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

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 1 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 2 illustrates the second step of the algorithm for the same example of Figure 1. 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 the step 1 are identified and CPPC can be used to perform the migration.

The performance of this mapping algorithm is assessed in next section.

3 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.

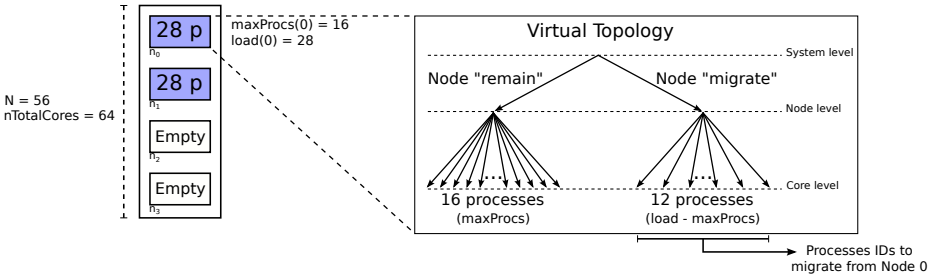


Fig. 1. Step 1: identifying processes to be migrated. Virtual topology built to migrate 12 processes from a 16-core node where 28 processes are running (16 processes remain and 12 processes migrate).

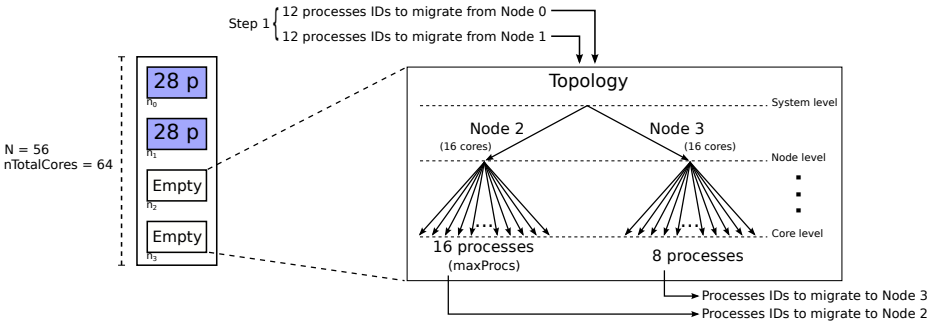


Fig. 2. Step 2: identifying target nodes. Topology built to map the migrating processes selected in step 1 to the empty cores in the system.

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 was composed of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [8] using class C. Only results for the FT benchmark are reported here due to space limitations. The FT benchmark presents the larger checkpoint files and the highest migration times of the NAS testbed. Additionally, the Himeno benchmark [5] 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 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.

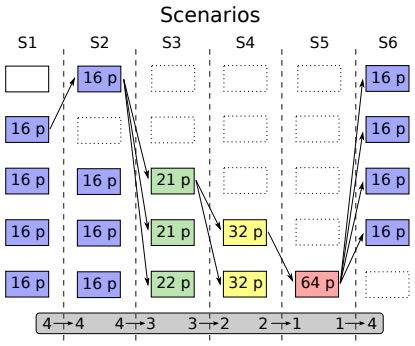
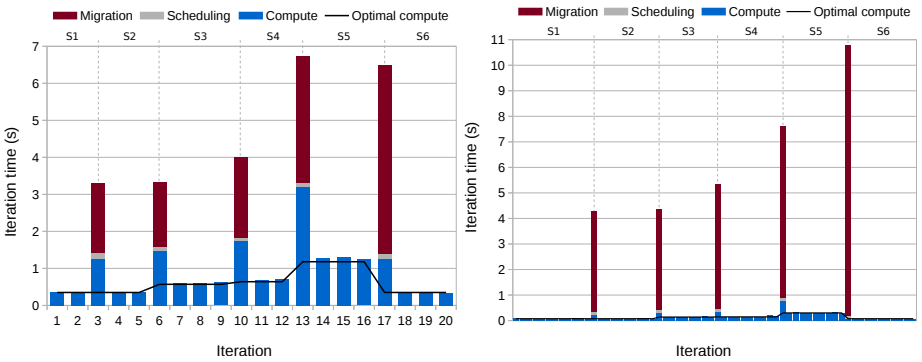


Fig. 3. Selected scenarios to show the feasibility and evaluate the migration and scheduling cost.

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 3 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 on the instantaneous performance impact.

Figure 4 shows the benchmarks results in the scenarios illustrated in Figure 3. These results demonstrate that, using the proposed solution, the applications are capable of adjusting the execution to changes in the environment. The high peaks in these figures correspond to reconfiguration points. The iteration time is broken down into: *scheduling*, *migration*, and *compute* times. The *scheduling* time is the execution time needed for the scheduling algorithm to identify processes to be moved and the target nodes. The *migration* time is the time needed to perform the migration itself. Finally, the *compute* time is the computational time of each iteration (excluding the *scheduling* and *migration* times when needed). As it can be seen, the *scheduling* time is negligible in comparison with the *migration* time, being always smaller than 0.1 s. The computational time of the iteration involved in the reconfiguration (*compute*), is higher than the computational time of the rest of the iterations. This is in part due to the cache misses caused



(a) FT (b) Himeno

Fig. 4. Iteration execution times in the scenarios illustrated in Figure 3

by the data movement, and in part due to an overhead observed in the first MPI communication after the communicators are reconfigured. For comparison purposes, the compute times that would be attained if the application could adjust its granularity to the available resources[4], instead of oversubscribing them without modifying the original number of processes are also shown (black line). The overhead that would introduce the data distribution needed to adjust the application granularity is not shown in the figure.

4 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 less than 1 second for the proposed scheduling algorithm, and of only a few seconds for the complete reconfiguration, which will be negligible in large applications with a 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.

[4] This time is measured executing the application with 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.)

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. A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.
2. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 180–186, Feb 2010.
3. W.L. Guay, S.A. Reinemo, B.D Johnsen, C.H Yen, T. Skeie, O. Lysne, and O. Tørudbakken. Early experiences with live migration of sr-iov enabled infiniband. *Journal of Parallel and Distributed Computing*, 78:39–52, 2015.
4. C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 306–322, 2003.
5. Information Technology Center, RIKEN. HIMENO Benchmark. `http://accc.riken.jp/2444.htm`. Last accessed November 2015.
6. E. Jeannot and G. Mercier. Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. In *Euro-Par*, pages 199–210, August 2010.
7. G. Martín, D.E. Singh, M.C. Marinescu, and J. Carretero. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, 46:60 – 77, 2015.
8. National Aeronautics and Space Administration. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. Last accessed July 2015.
9. A. Raveendran, T. Bicer, and G. Agrawal. A framework for elastic execution of existing MPI programs. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, pages 940–947, 2011.
10. F.S. Ribeiro, A.P. Nascimento, C. Boeres, V.E.F. Rebello, and A.C. Sena. Autonomic malleability in iterative mpi applications. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 192–199. IEEE, 2013.
11. M. Rodríguez, I. Cores, P. González, and M.J. Martín. Improving an MPI application-level migration approach through checkpoint file splitting. In *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 33–40, Paris, France, 2014.
12. S.S. Vadhiyar and J.J. Dongarra. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(02):291–312, 2003.
13. C. Wang, F. Mueller, C. Engelmann, and S.L. Scott. Proactive process-level live migration and back migration in HPC environments. *J. Parallel Distrib. Comput.*, 72(2):254–267, 2012.
14. D.B. Weatherly, D.K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *ACM/IEEE Conference on High Performance Networking and Computing (SC)*, pages 5–5, 2003.