Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI

Márcia C. Cera[1], Yiannis Georgiou[2], Olivier Richard[2], Nicolas Maillard[1], and Philippe O.A. Navaux[1]

[1] Universidade Federal do Rio Grande do Sul, Brazil
{marcia.cera,nicolas,navaux}@inf.ufrgs.br
[2] Laboratoire d'Informatique de Grenoble, France
{Yiannis.Georgiou,Olivier.Richard}@imag.fr

Abstract. Current parallel architectures take advantage of new hardware evolution, like the use of multicore machines in clusters and grids. The availability of such resources may also be dynamic. Therefore, some kind of adaptation is required by the applications and the resource manager to perform a good resource utilization. Malleable applications can provide a certain flexibility, adapting themselves on-the-fly, according to variations in the amount of available resources. However, to enable the execution of this kind of applications, some support from the resource manager is required, thus introducing important complexities like special allocation and scheduling policies. Under this context, we investigate some techniques to provide malleable behavior on MPI applications and the impact of this support upon a resource manager. Our study deals with two approaches to obtain malleability: dynamic CPUSETs mapping and dynamic MPI, using the OAR resource manager. The validation experiments were conducted upon Grid5000 platform. The testbed associates the charge of real workload traces and the execution of MPI benchmarks. Our results show that a dynamic approach using malleable jobs can lead to almost 25% of improvement in the resources utilization, when compared to a non-dynamic approach. Furthermore, the complexity of the malleability support, for the resource manager, seems to be overlapped by the improvement reached.

1 Introduction

Nowadays, a widely used trend for clusters of computers is to be composed by multicore machines. Furthermore, today's parallel architectures eventually include some dynamicity or flexibility in their resource availability. One example is the shared utilization of multicore machines. As applications have different execution times, the cores availability change dynamically. In such dynamic scenario, achieving a good resource utilization is a challenge. This paper studies alternatives to improve the resources utilization of current parallel architectures. Our motivations concern about *Quality of Service advantages* - better resources are used, faster work is executed and more users can be satisfied; *idle cycles minimization or elimination* - improvement of resources utilization can reduce idle cycles and conserve energy.

Improving resource utilization is directly related to the resource management and allocation. Resource management systems (RMS) are responsible to schedule jobs upon

K. Kant et al. (Eds.): ICDCN 2010, LNCS 5935, pp. 242–257, 2010.

the available resources and also to launch and monitor them. According to Feitelson and Rudolph [1], jobs can be: rigid, moldable, malleable or evolving. We are specially interested in malleable jobs because they can adapt themselves to resources with dynamic availability, and thus provide a better utilization of the current parallel architectures resources. However, malleable jobs demand a special treatment, with more complex allocation and scheduling policies. Thus, we investigate the complexity of treating malleable jobs and compare it with the gain/improvements of resources utilization.

The contribution of this paper is twofold: To study two different approaches that provide malleability to MPI jobs; To experiment those techniques upon a flexible resource management system in order to measure the advantages and the difficulties to support malleable jobs.

Our study uses two approaches to provide malleability: dynamic `CPUSETs` mapping and dynamic MPI. The first one is well-adapted to multicore machines and allows any parallel job to exploit malleability, through a fine control and manipulation of the available cores. The second approach uses a dynamic MPI application, able to adapt itself to the available cluster nodes. Note that, dynamic MPI means that applications are developed to be malleable employing MPI-2 dynamic process creation and fault tolerance mechanisms.

The impact of the support of these two malleability approaches upon resource managers, is evaluated considering a specific flexible RMS called OAR [2]. The experiments were performed upon Grid5000 platform using real workload traces of a known supercomputer. Our goal is to make a close analysis of the malleable application behavior in a scenario near to the real one. The results obtained show that cluster utilization can be improved in almost 25% using both of malleability approaches, when compared with a non-malleable approach. Furthermore, the advantages obtained by a possible integration of malleable jobs upon a resource manager seem to outperform the complexities of their support.

The remainder of this paper is organized as follows. Section 2 introduces the definitions and related works about job malleability and its support in today's resource managers. After that, Section 3 describes the two approaches to provide malleable behavior and the reasons of choosing them. Explanations about experimenting applications and their results, with `CPUSETs` mapping and dynamic MPI, are shown in Section 4. Finally, Section 5 describes our concluding remarks and the future work perspectives.

2 Related Works and Definitions

In the context of resource managers or job schedulers, parallel applications are represented as jobs, which have a running time t and a number of processors p. Feitelson and Rudolph [1] proposed four jobs classes based on the parameter p and the resource manager support. *Rigid* jobs require a fixed p, specified by the users, for a certain t time. In *moldable*, the resource manager choose p at start time from a range of choices given by the users. The job adapts to p that will not change at execution time. In *evolving*, p may change on-the-fly as job demand, which must be satisfied to avoid crashes. In *malleable*, p may also change on-the-fly, but the changes are required by the resource manager. In other words, malleable jobs are able to adapt themselves to changes in resources availability.

There are relevant theoretical results describing requirements, execution scheme and algorithms to efficiently schedule malleable applications [3]. Moreover, there is a growing interest to offer malleability in practice, because it can improve both resource utilization and response time [1,4,5,6,7]. Furthermore, a good approach towards the support of malleable jobs can be achieved by a co-design of runtime-system and programming environment [1]. This assertion can be observed in some practice malleability initiatives like PCM and Dynaco. PCM (Process Checkpointing and Migration) [8,9,10] uses migration to implement split and merge operations reconfiguring application according to resources availability. PCM demands an instrumentation of MPI code, using its API and specifying which data structures are involved in malleable operations. It interacts with a specific framework, IOS (Internet Operating System), performing like a resource manager, which is composed by an agents network to profile informations about resources and applications.

In the same way, Dynaco [11] enables MPI applications to spawn new processes when resources become available or stop them when resources are announced to disappear. To provide malleable features, Dynaco must know the adaptive decision procedures, a description of problems in the planning of adaptation, and the implementation of adaptive actions. Also, the programmer must include *points* on its source code identifying safe states to perform adaptive actions ensuring application correctness. Dynaco has its own resource manager, Koala, providing data and processor co-allocation, resource monitoring and fault tolerance.

Excluding initiatives with their own systems to provide resource management like the previously introduced, and according to our up-to-date knowledge, there is no existing implementation of malleable jobs support upon a generic resource manager since it is a rather complicated task. Nevertheless, some previous work have studied specific prototypes for the direct support of malleable jobs [7,6] with different constraints. For instance, Utrera et al. [6] proposes: a virtual malleability combining moldability (to decide the number of processes at start time), folding (to make the jobs malleable) and Co-Scheduling (to share resources). It uses Co-Scheduling to share processors after a job folding and migration to use the new processors after a job expansion. Our work also explores a similar folding technique, with dynamic `CPUSETs` mapping in multicore machines.

In [7], an Application Parallelism Manager (APM) provides dynamic online scheduling with malleable jobs upon shared memory architectures (SMP). The approach combined the advantages of time and space sharing scheduling. It achieved a 100% system utilization and a direct response time. Nevertheless the system was not directly adaptable to distributed-memory machines with message-passing and this work, as far as we know, has not evolved in a real implementation of the prototype.

We try to investigate the requirements of malleable jobs considering a generic resource management system, OAR [2]. It is an open source Resource Management System, which has evolved towards a certain 'versatility'. OAR provides a robust solution, used as a production system in various cases (Grid5000 [12], Ciment[1]). Moreover, due to its open architectural choices, based upon high level components (Database and Perl/Ruby Programming Languages), it can be easily extended to integrate new features

[1] https://ciment.ujf-grenoble.fr/cigri

and treat research issues. In our context, we exploit the capabilities of *Best Effort* jobs, also presented on [13]. They are defined as jobs with minimum priority, able to harness the idle resources of a cluster, but they will be directly killed when the resources are required. The idea of *Best Effort* jobs is comparable with the notion of 'cycle stealing' initiated by Condor [14] and the High Throughput Computing approach.

3 Supporting Malleability with Dynamic `CPUSETs` Mapping and Dynamic MPI

Based on issues highlighted in Section 2, we study the use of malleability to improve the utilization of current parallel architectures. Through our experiences, we evaluate the resources utilization and the overhead to support malleability on MPI applications. Under our context, the resources dynamicity is represented by their availability. In other words, once it is possible identify the unused resources (*i.e.* the idle ones), they can be used in low priority executions. Such set of unused resources will be change on-the-fly according new jobs arrives or old ones finishes characterizing its dynamicity.

Dynamic `CPUSETs` mapping and dynamic MPI were the approaches selected by us to provide malleability. As we explain later they have different levels of complexity. Our goal is to experiment those techniques upon a resource management system, in order to be able to value the advantages and the complexities of their support. We therefore constructed a prototype using features of OAR resource management system. OAR modularity and flexibility permitted us to integrate the above techniques without actually implementing them on its core.

The module that we developed aside OAR is based on two notions: the *Best Effort* job type as described in Section 2 and a resource discovery command, which provides the current and near-future available resources. To provide malleability using OAR we consider that a malleable job is constructed by a rigid and a *Best Effort* part. The rigid part stays always intact so that it can guarantee the job completeness. In the same time, the *Best Effort* part is responsible for the flexibility of the job. Hence, using the resource discovery command, which informs the application for the variations on the resources availability, we come to two scenarios: *Best Effort* jobs can be either *killed* meaning malleable job shrinking or further *submitted*, allowing malleable job growing. This prototype based on bash scripts is transparent to the user and does not request special care about the two parts of the OAR malleable job, in any of the two proposed malleability techniques.

3.1 Dynamic `CPUSETs` Mapping Requirements

`CPUSETs`[2] are lightweight objects in the Linux kernel that enable users to partition their multiprocessor machine by creating execution areas. `CPUSETs` are used by the resource management systems to provide CPU execution affinity and cleaner process deletion when the job is finished.

Dynamic `CPUSETs` mapping technique is the on-the-fly manipulation of the amount of cores per node allocated to an application. This technique is simple and well-adapted

[2] http://www.bullopensource.org/cpuset/

to multicore machines. CPUSETs mapping allows expanding or folding of the application upon the same node. In more detail, if we have multiple processes of a MPI application executing upon a multicore node, then those processes can be executed upon one or more cores of the same node. CPUSETs mapping is a system level technique and it is transparent to the application. It can provide a level of malleability upon any MPI application and it is completely independent of the application source code. The only restriction is that the malleability is performed separately upon each node, which means that multiple processes have to be initiated upon at least one core of a node. Furthermore to take full advantage of the malleability possibilities, the number of processes started upon a node should be equal or larger than the number of cores per node. For instance, a 16 processes MPI application should be ideally initiated upon 4 nodes of a 4 cores-per-node cluster or upon 2 nodes of a 8 cores-per-node cluster.

In the context of CPUSETs mapping technique, the specific OAR malleability prototype, defines the rigid part of a malleable job to occupy one core of each participating node. This feature comes from the CPUSETs restriction explained above. When more cores of the same nodes become available, then *Best Effort* jobs are further submitted. In the same time the CPUSETs mapping technique performs the expanding operation so that the application processes that were sharing a core migrate to the newly available cores. In the opposite case, when an external rigid job asks for resources, some *Best Effort* jobs need to be killed. In this case, the CPUSETs mapping technique performs a folding of processes on the fewer remaining cores, and the demanded cores are given to the arriving rigid job. Therefore, malleability is achieved by the use of *Best Effort* jobs and the CPUSETs folding and expanding of processes. Figure 1 presents some scenarios, showing the different stages for only one of the participating nodes of the cluster.

It seems that besides the restrictions, this system level approach can actually provide malleability without complicating the function of OAR resource manager. Nevertheless, there are issues that have to be taken into account. The overhead of the expanding or folding operation upon the application has to be measured. Furthermore, since our context concerns cluster of shared memory architectures, it will be interesting to see how two different MPI applications running upon different cores on the same node, would perform during the expanding and folding phases.

3.2 Dynamic MPI Requirements

Dynamic MPI applications have malleable behavior since they perform growing and shrinking operations adapting themselves to variations in the availability of the resources. In growing, some application workload is destined to new resources through MPI processes spawning, which is a feature of the MPI-2 norm (MPI_Comm_spawn and correlated primitives) [15]. In shrinking, processes running on resources announced as unavailable must be safely finalized and some procedure is required to prevent application crash. We adopted a simple one: tasks executing in processes being shrunk are identified and will be restarted in the future. It is not optimal, but ensures application results correctness.

The malleability operations, growing and shrinking, are handled by a library called *lib-dynamicMPI*. This library interact with the OAR to receive information about

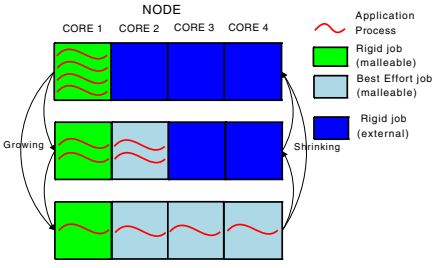

Fig. 1. Behavior of `CPUSETs` mapping technique upon one of the participating nodes

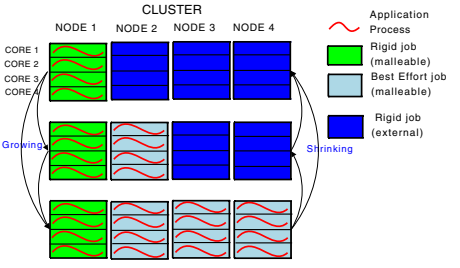

Fig. 2. Dynamic MPI application performing growing and shrinking upon 4 participating nodes

variations in the amount of resources available to dynamic MPI applications. According to these informations, *lib-dynamicMPI* launches the appropriate malleable action enabling the adaptation of the application to the availability of the resources, and thus the library is like a bridge between MPI applications and resource manager. As part of the growing operation, the *lib-dynamicMPI* ensures that spawning processes will be placed into the new resources. The library intercepts the `MPI_Comm_spawn` calls and set the physical location of the new processes according to the informations provided by the OAR. This feature is an evolution of our previous work, in which the destination of the dynamic processes was decided following one of two policies: Round-Robin (standard) and workload-based (to chose the less overloaded resource) [16], without interactions with a resource manager.

In the experiments of this paper, *lib-dynamicMPI* and dynamic MPI application are implemented with LAM/MPI[3] distribution. This MPI distribution offers a stable implementation of dynamic process creation and ways to manage dynamic resources. This last feature is provided by two commands: `lamgrow` to increase and `lamshrink` to decrease the amount of nodes available in LAM/MPI network (LAM/MPI applications run up a known set of resources, which begins by `lamboot` before application starting time and ends by `lamhalt` after application execution). Note that LAM/MPI enables the management of nodes, which can be composed by many cores, but cannot manage isolated cores. The `lamgrow` and `lamshrink` commands will always be called by *lib-dynamicMPI* when some change is announced by OAR. Once there are changes in LAM/MPI network, the malleable applications perform the appropriate operation - growing or shrinking.

As previously explained, in our experiments, a malleable job is composed by rigid and *Best Effort* parts. In the context of dynamic MPI, the rigid part is composed by the minimum unit manageable in LAM/MPI, *i.e.* one node. The *Best Effort* part is as large as the amount of resources available, which is determined by the resource discovery command. The malleable application begins by taking into account the initial amount of resources available, *i.e.*, the initial size of the malleable job. When a *Best Effort* job is further submitted, *lib-dynamicMPI* is notified and launches the `lamgrow`. As

[3] http://www.lam-mpi.org/

a result of LAM/MPI network increase, a growing operation is performed on the MPI application. In the opposite case, when some nodes are demanded to satisfy arriving jobs, they will be requested from the *Best Effort* part. The *lib-dynamicMPI* is notified and performs the `lamshrink`, launching the shrinking operation and the procedure to finalize the execution of the processes placed into demanded resources. To ensure that this operation will be safely performed, a grace time delay is applied upon OAR, before the killing of the *Best Effort* jobs [13]. Grace time delay represents the amount of the time that the OAR system waits before destinate the resources to another job, ensuring that they are free. Figure 2 illustrates a malleable job upon 4 nodes with 4 cores performing growing and shrinking operations.

4 Experimental Evaluation

This section presents results using resources from Grid5000 platform [12]. The experiments were performed upon a cluster (IBM System x3455) composed by DualCPU-DualCore AMD Opteron 2218 (2.6 GHz/2 MB L2 cache/800 MHz) and 4GB memory per node with Gigabit Ethernet.

We chose two applications to investigate the malleable jobs requirements. For `CPUSETs` mapping technique we used the NAS benchmarks [17], which are widely used to evaluate the performance of parallel supercomputers. We will present results of BT (Block Tridiagonal Solver) and CG (Conjugate Gradient) class C benchmarks with their MPI3.3 implementation[4]. Although our approach is applicable to the entire set of NAS applications, we chose the results of two of them to details in this paper for space reasons.

In dynamic MPI tests, we used a malleable implementation of the Mandelbrot fractal *i.e.* a Mandelbrot able to grow or shrink, during the execution time, according the resource availability. Although Mandelbrot problem does not require malleable features to be solved, we adopted a well-known MPI application to become malleable through *lib-dynamicMPI*. The malleable Mandelbrot starts with a master that will spawn workers (one by core available), manager tasks and store the results. Workers receive tasks, execute them, return results and wait for more tasks. Mandelbrot is able to identify resources changes and launch the malleable operations. In growing, new workers are spawned upon resources that become available, and in shrinking workers inform to master which are the executing tasks and then finalize their execution.

Two series of experiments were chosen to test malleability through the above 2 techniques. The first operates through increasing or decreasing of resources, without using the resource manager. In this way, we can evaluate the malleability impact in applications performance without considering the issues related to the resource manager. The results of these experiments are presented in Sections 4.1 and 4.2.

The second series of experiments evaluates the OAR resource manager support to malleable jobs. Our testbed is based on workload trace injection, to provide the typical rigid jobs of the cluster, along with a demo that submits one malleable job at a time. In such initial work we do not deal with multiple malleable jobs which initiate more complex scheduling issues (equipartitioning, etc). We measure the overall resources

[4] http://www.nas.nasa.gov/Resources/Software/npb_changes.html

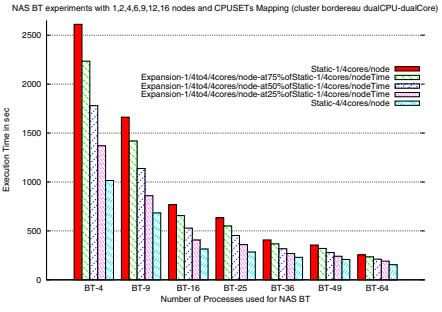

Fig. 3. NAS BT-(4,9,16,25,36,49,64) behavior with Static and Dynamic CPUSETs mapping operations with direct expansion from 1 to 4 cores

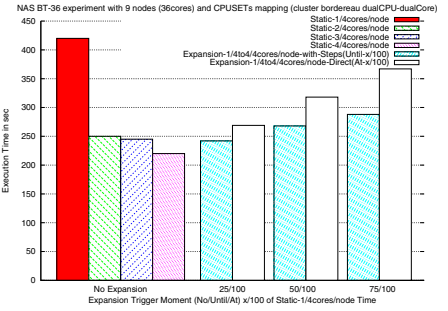

Fig. 4. NAS BT-36 behavior with Static and Dynamic CPUSETs mapping operation with gradual and direct expansion from 1 to 4 cores

utilization and the impact of malleability upon the rigid workload. In more detail, the experiments are based on real workload traces from DAS2 grid platform [18], which represent the workload of twelve month (year 2003) on 5 clusters. A specific cluster and time range is selected from these traces, to charge Grid5000 resources. Malleable applications will operate using the idle resources, *i.e.* the resources that were not charged from normal workload injection. In this way, the dynamic events are guided by resource availability as can be seen in Section 4.3.

4.1 Malleability on Multicore Nodes with Dynamic CPUSETs Mapping

In this series of experiments we evaluate the impact of the CPUSETs mapping technique upon NAS benchmarks. In Figure 3 we present the results of NAS BT application performing static and dynamic CPUSETs mapping. In each case the 'Static-1/4cores/node' box implies the use of 1 core per participating node that is the minimum folding of processes that can be applied. On the other hand, 'Static-4/4cores/node' box implies the use of 4 cores per node, which represents the maximum expansion of processes that can be made upon a dualCPU/dualCore architecture. For instance, in the case of BT-36 we use 9 nodes with 4 processes running upon each node: 4 processes on 1 core for the Static-1/4 case and 1 process on 1 core for the Static 4/4 case. The 3 boxes between the above 'Static-1/4' and 'Static-4/4' instances represent different dynamic instances using the dynamic CPUSETs mapping approach. All 3 instances imply the use of 1 core per participating node at the beginning of the execution performing a dynamic CPUSETs expansion to use 4 cores after specific time intervals. The expansion trigger moment is placed on the 25%, 50% and 75% of the 'Static-1/4cores/node' execution time of each BT.

It is interesting to see the speedup between the Static cases of '1/4' and '4/4' cores/node among each different BT. We observe that this speedup becomes more important as the number of BT processes go smaller. In the case of BT-4 the speedup from the use of 4/4 cores/node to 1/4 is 1597 sec. which means 61,5% of the 1/4 cores/node time. Whereas the relevant speedup of BT-25 is 350 sec. or 53,8% of the 1/4 time, and to

BT-64 is 100 sec. or 33,3% of the 1/4 cores/node time. Furthermore, the figure shows that the technique of dynamic CPUSETs mapping expansion from 1 to 4 cores works fine without complicated behavior. Linear increase of the expansion trigger moment results in a linear increase of the execution time.

Figure 4 illustrates the behavior of NAS BT-36 execution upon 9 nodes. The figure shows Static and Dynamic CPUSETs mapping executions along with direct and gradual expansion cases. Considering the static cases (with no expansions during the execution time), we observe a really marginal speedup between the 2/4 to 3/4 and 3/4 to 4/4 cores/node cases (5 sec. and 25 sec. respectively) in contrast with an important speedup between the 1/4 to 2/4 cases (200 sec.). This could be partially explained by a communication congestion for BT after using more than 2cores/node. The histograms representing the dynamic CPUSETs direct and gradual expansions, show us that gradual expansions result in faster execution times. Indeed for each case of the gradual expansion, we perform 3 expansions: 1/4 to 2/4, 2/4 to 3/4 and 3/4 to 4/4 cores/node until a specific instant. On the opposite case of direct expansion we perform only one: from 1/4 to 4/4 cores/node at the above same instant. Thus, after this same instant, both cases are using 4/4 cores/node. The difference is that in the first one there has already taken place 3 expansions whereas in the second one only 1 expansion. For BT benchmark, the results show that it is better to perform gradual small expansions whenever resources are free, in contrast of waiting multiple free resources and perform one bigger direct expansion. Moreover, these results show that the overhead of the expansion process, is marginal, when trading-off with the gain of using the one or more new cores.

Figure 5 illustrates the experiments of 3 different cases of BT upon different number of nodes. In more detail we try to measure the impact of the use of different number of nodes - with one process per core - on their execution time. For every different BT, we experiment with the ideal case of using equal number of nodes with processes (BT-16,36,64 with 16,36 and 64 nodes respectively) along with the case of using equal number of processes with cores (BT-16,36,64 with 4,9 and 16 nodes respectively). It seems that it is always faster to use more nodes. This can be explained by the fact that processes do not share communication links and memory. Nevertheless, this speedup is not very significant implying that, for BT application, the second case is more interesting to use. This is because, we can also perform malleability operations with dynamic CPUSETs mapping techniques. Hence, although we have a worst performance we can achieve better overall resources utilization.

Figure 6 presents static and dynamic cases in a Dedicated (DM) or Shared Memory (SM) context of BT-64 and CG-64 benchmarks. Concerning the Static and Dedicated memory cases of BT-64 and CG-64 we can observe that the only important speedup is between the use of 1/4 to 2/4 cores/node for BT-64, whereas for all the rest the performance is quite the same. In the shared memory context, all performances decreased as expected because the benchmarks were sharing memory and communication links. Nevertheless, BT-64 presented an important sensitivity to resources dynamism. This can be observed by the fact that we have a significant performance degradation for BT-64 when it uses 1 core and CG-64 uses 3 cores of the same node; and by the fact that the performance is upgraded for BT-64 when the above roles are inversed. Finally it seems that a dynamic on-the-fly expansion with CPUSETs mapping technique, using

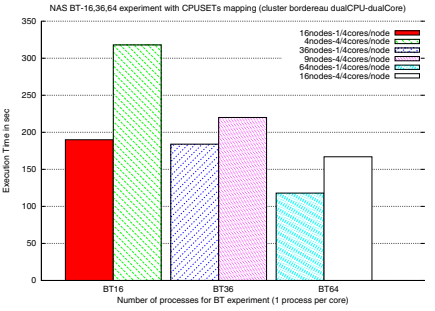

Fig. 5. NAS (BT-16,36,64) behavior with 1 process per core, different number of nodes and Static `CPUSETs` mapping operations

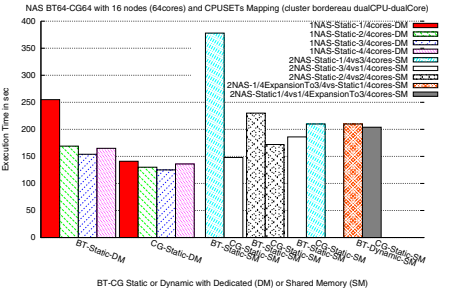

Fig. 6. NAS BT-64 and CG-64 behavior, with Shared Memory and Static/Dynamic `CPUSETs` mapping operations

from 1 to 3 cores/node, achieves a good performance as compared to the Static cases with shared memory. On the same time the degradation of CG-64 performance is small when BT-64 is expanding.

In the second series of our experiments we will be using BT and CG in a shared memory context. According to those last results, where BT presents more sensitivity in resources dynamism than CG, we decided to implicate BT with malleability whereas CG will represent the rigid applications. We have conducted experiments using all different NAS benchmarks and the `CPUSETs` Mapping technique was validated in all cases. Nevertheless, different performances were observed. Future extended versions of this work will include these experiments.

The figures of this section show the results obtained when performing only expanding operations. Nevertheless, our experiments indicated that the behavior is similar when performing folding operations. Due to lack of space, they were omitted. An in-depth analysis of the memory contention in a `CPUSETs` mapping context is currently under study and will be discussed and quantified in future works.

4.2 Malleability on a Cluster with Dynamic MPI

In the experiments of this section, we used the maximum of 64 cores, coming from 16 nodes (dualCPU/dualcore per node). Figures 7 and 8, show the execution time of the dynamic MPI application performing growing and shrinking operations respectively. In the growing, the application begins with the amount of cores shown in x axis (representing 25%, 50% and 75% of the total number of cores) and grows until 64 cores. In shrinking, it starts with 64 and shrinks until the x values of cores. We made two experiments: *(i)* a dynamic event is performed **at** a time limit, and *(ii)* dynamic events are gradually performed **until** a time limit. Time limit is defined as 25%, 50% or 75% of the parallel reference time. Reference time is always the execution time with the initial number of cores, *i.e.*, x to growing and 64 to shrinking.

Let p_r be the number of cores on which a reference time t_r has been measured. Thus, the parallel reference application performs $W = t_r \times p_r$ operations. The application is progressively run on more than p_r cores:

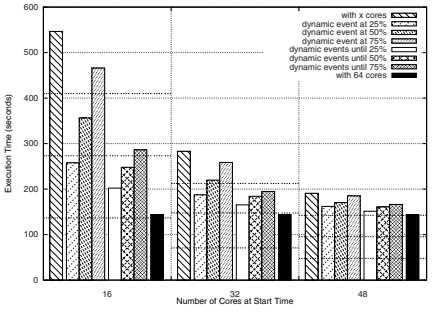

Fig. 7. Growing in dynamic MPI application: execution time *vs.* number of cores at starting time

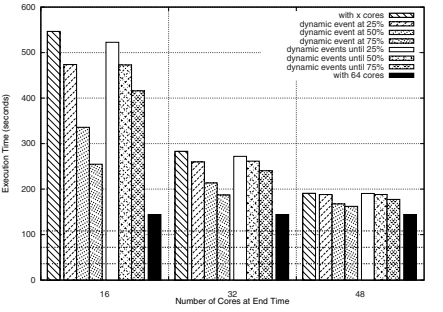

Fig. 8. Shrinking in dynamic MPI application: execution time *vs.* number of cores at ending time

- In the first experiment (dynamic events at a time limit) it is run during $\alpha t_r (0 \leq \alpha \leq 1)$ seconds on p_r cores, and then on p cores;
- In the second experiment (dynamic events until a time limit) it is run during αt_r $(0 \leq \alpha \leq 1)$ seconds on a gradual increasing or decreasing number of cores, until reaching p cores. Then it is run until completion, without anymore changes. The number of cores increases or decreases at regular timesteps of c units. In fact, $c = 4$, since the `lamgrow` command only enables to add a whole node (2 CPUs with 2 cores) to LAM/MPI network. The number of gradual timesteps is therefore $\frac{p-p_r}{c}$, and each one lasts $\delta = \frac{\alpha t_r}{\frac{p-p_r}{c}}$ sec.

Ideal speedup in the first experiment. The execution on p_r cores has duration αt_r. At this point, $(1-\alpha)W$ operations remain to be performed, by p cores. Ideally, this second phase runs in time $(1-\alpha)W/p$. Therefore, the total parallel time in this case is $t_p = \alpha t_r + (1-\alpha)t_r p_r/p$, and the speedup t_r/t_p is:

$$S = \frac{1}{\frac{p_r}{p}(1-\alpha)+\alpha} \tag{1}$$

Ideal speedup in the second experiment. As in the previous case, the number of operations performed during the first αt_r seconds, can be computed, to obtain the parallel time of the second part of the execution on p cores. At the ith timestep ($i = 0 \ldots \frac{p-p_r}{c} - 1$), the program is run with $p_r + ci$ cores in time δ sec. Therefore, the number of operations n_i that are executed is $\delta(p_r + ci)$. When all the p cores are available (*i.e.* at time αt_r), $\sum_{i=0}^{(p-p_r)/c-1} n_i$ operations have been run, leaving $W - \sum_{i=0}^{(p-p_r)/c-1} n_i$ to be run by p cores. Thus, the second phase has duration:

$$\frac{t_r p_r - \sum_{i=0}^{(p-p_r)/c-1} \delta(p_r + ci)}{p}, \tag{2}$$

and the total execution time in this second experiment is therefore:

$$t_p = \alpha t_r + \frac{t_r p_r - \sum_{i=0}^{(p-p_r)/c-1} \delta(p_r + ci)}{p}. \tag{3}$$

Besides,

$$\sum_{i=0}^{(p-p_r)/c-1} \delta(p_r + ci) = \delta p_r \frac{p-p_r}{c} + c\delta \sum_{i=0}^{\frac{p-p_r}{c}-1} i = \delta p_r \frac{p-p_r}{c} + c\delta \frac{(\frac{p-p_r}{c}-1)\frac{p-p_r}{c}}{2}. \tag{4}$$

And since $\delta = \frac{\alpha t_r}{\frac{p-p_r}{c}}$, the latter equation yields:

$$\sum_{i=0}^{(p-p_r)/c-1} \delta(p_r + ci) = \alpha t_r p_r + \frac{\alpha t_r}{2}(p - p_r - c). \tag{5}$$

Therefore, the total parallel time in this case is:

$$t_p = \alpha t_r + \frac{t_r p_r - \alpha p_r t_r - \frac{\alpha t_r}{2}(p - p_r - c)}{p} = \frac{t_r}{2p}(2p_r + \alpha(p - p_r + c)). \tag{6}$$

And the parallel speedup t_r/t_p is:

$$S = \frac{2p}{2p_r + \alpha(p - p_r + c)}. \tag{7}$$

Table 1. Speedup of the dynamic MPI application

Growing							Shrinking						
p_r	p	α	S	$S(eq.1)$	S	$S(eq.7)$	p_r	p	α	S	$S(eq.1)$	S	$S(eq.7)$
16	64	0.25	2.12	2.29	2.70	2.84	64	16	0.25	0.30	0.31	0.28	0.27
		0.50	1.53	1.60	2.20	2.21			0.50	0.43	0.40	0.30	0.30
		0.75	1.17	1.23	1.91	1.80			0.75	0.57	0.57	0.35	0.34
32	64	0.25	1.51	1.60	1.71	1.75	64	32	0.25	0.55	0.57	0.53	0.53
		0.50	1.29	1.33	1.54	1.56			0.50	0.68	0.67	0.55	0.56
		0.75	1.09	1.14	1.45	1.41			0.75	0.77	0.80	0.60	0.60
48	64	0.25	1.18	1.23	1.26	1.27	64	48	0.25	0.77	0.80	0.76	0.77
		0.50	1.12	1.14	1.18	1.21			0.50	0.86	0.86	0.77	0.79
		0.75	1.03	1.07	1.15	1.15			0.75	0.89	0.92	0.81	0.81

Table 1 shows the speedups, in which S is the speedup in practice (t_r/t_p), $S(eq.1)$ is the ideal speedup to the first experiment using the Equation 1, and $S(eq.7)$ to the second one using the Equation 7. In growing, practical speedups are slightly lower than the ideal ones, representing the overhead to perform the spawning of new processes. As Lepère et al. [3] the standard behavior in on-the-fly resources addition is that such addition cannot increases the application execution time. In the Table 1, we observe that growing speedups are always greater than 1, meaning that the new cores could improve application performance decreasing its execution time as expected. In shrinking, all speedups values are lower than 1, as the exclusion of nodes decreases the performance. In this case, practical speedup is quite similar to the theoretical one and the variations become from the execution of the procedures to avoid application crash. Summing up, the speedups show that our dynamic MPI application is able to perform upon resources with a dynamic availability.

4.3 Improving Resource Utilization Using Malleability

The second series of experiments, aims at the evaluation of the two malleability techniques upon the OAR resource manager. Experiments for both approaches, follow the same guidelines as explained in Section 4. A workload of 5 hours 'slice' of DAS2 workload with 40% of cluster utilization is injected into OAR. This workload charges the resources, representing the normal workload of the cluster. At same time one malleable job per time is submitted and will run upon the free resources, *i.e.* those that are not used by the normal workload.

The main differences of the experiments, for the two different malleability techniques, lay upon the type of application executed and on the way the malleable job is submitted on each case. For the `CPUSETs` mapping approach we execute BT benchmarks for the malleable job and CG benchmarks for the static workload jobs, so that we can observe the impact on resources utilization in a shared memory context. The malleable job is submitted to OAR in accordance to the guidelines of Section 3.1. This means that one core per participating node has to be occupied by the rigid part of the malleable job. Since the time of one NAS BT execution is rather small, especially with big number of processes, we decided that the malleable job will have to execute 8 NAS BT applications in a row. At the same time, CG benchmarks are continuously executed during the normal jobs allocation, as noted in the workload trace file. The number of processes for each NAS execution during the experiment is chosen according to the number of current free resources.

In the experiments of the Dynamic MPI case, the malleable job implies the execution of Mandelbrot benchmark. The normal workload jobs (from DAS2 workload traces) are empty 'sleep' jobs, just occupying the resources for a specific time. Since the malleability is performed using whole nodes, there was no need to perform real application executions with the normal workload jobs. As explained on subsection 3.2, the malleable job is submitted to OAR, by occupying only one whole node in the rigid part of the malleable job and the *Best Effort* part is as large as the amount of remaining resources.

The dynamic executions of both techniques are compared with non-dynamic experiments running the same applications. In more details, the malleable jobs submission is substituted by moldable-besteffort jobs submission. As moldable-besteffort job, we define a moldable job that starts its execution upon all free cluster resources and remains without changes until its execution ending. This means that if more resources become available they will be not used by the moldable-besteffort job and when some resources are demanded to supply arriving jobs, it will be immediately killed, like a *Best Effort* job.

Since the experimental results figures are very similar for both malleability approaches, we include only the figures featuring the support of `CPUSETs` mapping technique upon OAR. Figures 9 and 10 show the results of malleable and moldable-besteffort jobs respectively. It is interesting to observe the gain upon the cluster resources utilization to the dynamic context, presented in Figure 9, as compared to the non-dynamic case in Figure 10. The white vertical lines of Figure 9 until 5500 sec., represent the release of *Best Effort* jobs resources. This happens, when one group of 8 BT executions are finished until the next malleable job begins and new *Best Effort* jobs occupy the free resources. Also, at starting time the malleable job begins, only with the

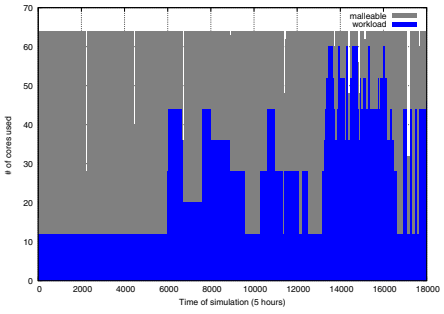

Fig. 9. Malleable job executing BT application upon the free resources of the normal workload

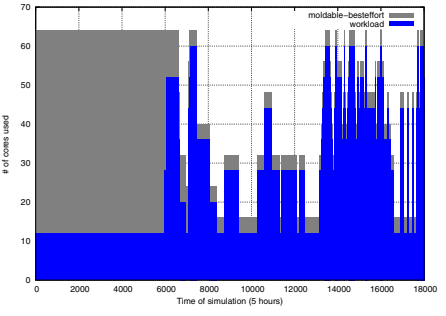

Fig. 10. Moldable-besteffort job executing BT application upon the free resources of the normal workload

rigid part (*i.e.* only one core) and immediately expands using the other available cores in the *Best Effort* part of the malleable job. After 5500 sec. the execution of malleable jobs start to be influenced by jobs from the workload (normal jobs). In that way, the white lines (empty spaces) also mean that the application did not have enough time to grow itself, before a new normal job arrived.

In terms of resources utilisation, since the normal workload makes use of 40% of cluster resources, this leaves a total 60% of free resources. Moldable-besteffort jobs use a 32% of the idle resources arriving at 72% of total cluster resources used. On the other hand, in the dynamic context of malleability approaches, the use of idle resources reach 57% arriving at 97% of overall cluster utilisation. Hence, an improvement of almost **25%** of resources utilization is achieved when the dynamic approaches are compared with the non-dynamic one. Furthermore, we observed the number of jobs executed during the 5 hours of experimentation. In the dynamic context, we obtained 8 successfully 'Terminated' malleable jobs, compared to 4 'Terminated' and 5 in 'Error State', for the non-dynamic one. The impact of the response time, for the normal workload, was also measured. The results have shown 8 sec. of average response time in the non-dynamic context, compared to 81 sec. of average response time in the dynamic `CPUSETs` mapping technique and 44 sec. in the Dynamic MPI technique. The big response time of the dynamic `CPUSETs` mapping technique, is explained by the allocation of one core per node by the rigid part of the malleable job. This limits the number of free resources for jobs coming from the normal workload. The response time for Dynamic MPI approach, is explained by the 40 sec. grace time delay to the MPI malleable application. This grace time was added to OAR for the shrinking operations, as presented on Section 3.2.

Although our initial experiments consider only one workload trace and one malleable job at a time, they enabled the verification of a significant improvement in resource utilization brought by the employment of the malleable jobs when compared to a non-dynamic approach. In other words, we reach our experiments goal, which was verify if the efforts to enable malleable jobs upon OAR would be outperformed by the gain reached in resource utilization. Future works will include experiments with different workload traces and multiple malleable jobs.

5 Conclusions and Future Works

In this paper we aimed to employ malleability features to provide better resources utilization. Furthermore, we analyzed the impact of the malleability support upon a cluster resource manager. In our context, the resources dynamicity is guide by their availability, *i.e.*, unused (or idle) resources are used to perform malleable jobs, improving the resource utilization. Our study focused on dynamic `CPUSETs` mapping and dynamic MPI as ways to provide malleability. Both techniques were implemented as a prototype upon OAR resource manager. The approach of dynamic `CPUSETs` mapping applied to provide malleability, upon OAR resource manager is - as far as we know - a new idea in the area. It is an interesting alternative to perform a level of malleable operations efficiently, which does not require changes in the application source code. On the other hand, some constraints must be respected: *i.e.* always start processes upon at least one core of all participating nodes and have as much processes as the maximum amount of cores that can be available. We also identified that besides `CPUSETs` mapping technique, a good memory management is essential to memory-bound applications. In the future, we wish to improve our `CPUSETs` study considering memory issues, by increasing the range of applications which perform well using this technique.

The implementation of MPI-2 features in current MPI distributions, gives the means to develop malleable applications. The impact of malleability upon MPI applications, is outperformed by the gain upon resources utilization. On the other hand, the applications should be notified with the resources availability. Such information is provided to the MPI applications from the OAR. Moreover, we believe that some support to resources dynamism from MPI distributions is required by malleable applications, as provided in LAM/MPI by `lamgrow` and `lamshrink`. Our future work will aim to the *lib-dynamicMPI* independence from MPI distribution. We also consider to increase the range of programs, to study the malleability upon non-bag-of-tasks applications. Finally, some improvements upon communication between OAR and *lib-dynamicMPI*, have to be made.

The study of the two malleability approaches - dynamic `CPUSETs` mapping which is in system level and dynamic MPI application which is in application level - upon a resource manager is another important contribution of this paper. We identified both approaches requirements and their support by an OAR module and we tried to relate their impact on complexity and gain. We concluded that the complexity is outperformed by the gain. This conclusion, based on tests - using a real workload injected in OAR - proved that malleable jobs as compared to a non-malleable approach, enable an almost **25%** greater resource utilization.

Acknowledgements

The authors would like to thank CAPES, Bull OpenSource and MESCAL project for supporting this research. Furthermore, we thank Grid5000 for their experimental platform.

References

1. Feitelson, D.G., Rudolph, L.: Toward convergence in job schedulers for parallel supercomputers. In: Job Scheduling Strategies for Parallel Processing, pp. 1–26. Springer, Heidelberg (1996)
2. Capit, N., Costa, G.D., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: 5th Int. Symposium on Cluster Computing and the Grid, Cardiff, UK, pp. 776–783. IEEE, Los Alamitos (2005)
3. Lepère, R., Trystram, D., Woeginger, G.J.: Approximation algorithms for scheduling malleable tasks under precedence constraints. International Journal of Foundations of Computer Science 13(4), 613–627 (2002)
4. Kalé, L.V., Kumar, S., DeSouza, J.: A malleable-job system for timeshared parallel machines. In: 2nd Int. Symposium on Cluster Computing and the Grid, Washington, USA, pp. 230–238. IEEE, Los Alamitos (2002)
5. Hungershöfer, J.: On the combined scheduling of malleable and rigid jobs. In: 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 206–213 (2004)
6. Utrera, G., Corbalán, J., Labarta, J.: Implementing malleability on MPI jobs. In: 13th Int. Conference on Parallel Architectures and Compilation Techniques, pp. 215–224. IEEE, Los Alamitos (2004)
7. Hungershöfer, J., Achim Streit, J.M.W.: Efficient resource management for malleable applications. Technical Report TR-003-01, Paderborn Center for Parallel Computing (2001)
8. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Malleable iterative mpi applications. Concurrency and Computation: Practice and Experience 21(3), 393–413 (2009)
9. Maghraoui, K.E., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic malleability in iterative MPI applications. In: 7th Int. Symposium on Cluster Computing and the Grid, pp. 591–598. IEEE, Los Alamitos (2007)
10. Desell, T., Maghraoui, K.E., Varela, C.A.: Malleable applications for scalable high performance computing. Cluster Computing 10(3), 323–337 (2007)
11. Buisson, J., Sonmez, O., Mohamed, H., Epema, D.: Scheduling malleable applications in multicluster systems. In: Int. Conference on Cluster Computing, pp. 372–381. IEEE, Los Alamitos (2007)
12. Bolze, R., Cappello, F., Caron, E., Dayd, M., Desprez, F., Jeannot, E., Jgou, Y., Lantri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, l.G., Ira, T.: Grid 5000: a large scale and highly reconfigurable experimental grid testbed. Int. Journal of High Performance Computing Applications 20(4), 481–494 (2006)
13. Georgiou, Y., Richard, O., Capit, N.: Evaluations of the lightweight grid cigri upon the grid 5000 platform. In: Third IEEE International Conference on e-Science and Grid Computing, Washington, DC, USA, pp. 279–286. IEEE Computer Society, Los Alamitos (2007)
14. Litzkow, M., Livny, M., Mutka, M.: Condor - a hunter of idle workstations. In: Proceedings of the 8th International Conference of Distributed Computing Systems (1988)
15. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2 Advanced Features of the Message-Passing Interface. The MIT Press, Cambridge (1999)
16. Cera, M.C., Pezzi, G.P., Mathias, E.N., Maillard, N., Navaux, P.O.A.: Improving the dynamic creation of processes in MPI-2. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 247–255. Springer, Heidelberg (2006)
17. Bailey, D., Harris, T., Saphir, W., Wijngaart, R.V.D., Woo, A., Yarrow, M.: The nas parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center (1995)
18. Li, H., Groep, D.L., Wolters, L.: Workload characteristics of a multi-cluster supercomputer. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 176–193. Springer, Heidelberg (2005)