Hybrid Preemptive Scheduling of MPI Applications on the Grids

Aurélien Bouteiller, Hinde-Lilia Bouziane, Thomas Herault, Pierre Lemarinier, Franck Cappello
INRIA/LRI, Université Paris-Sud, Orsay, France
E-mail: { bouteill, bouziane, herault, lemarini, fci }@lri.fr
phone: (+33) 1 69 15 4222
fax: (+33) 1 69 15 4213

Abstract— **Time sharing between cluster resources in Grid is a major issue in cluster and Grid integration. Classical Grid architecture involves a higher level scheduler which submits non overlapping jobs to the independent batch schedulers of each cluster of the Grid. The sequentiality induced by this approach does not fit with the expected number of users and job heterogeneity of the Grids. Time sharing techniques address this issue by allowing simultaneous executions of many applications on the same resources.**

Co-scheduling and gang scheduling are the two best known techniques for time sharing cluster resources. Co-scheduling relies on the operating system of each node to schedule the processes of every application. Gang scheduling ensures that the same application is scheduled on all nodes simultaneously. Previous work has proven that co-scheduling techniques outperforms gang scheduling when physical memory is not exhausted.

In this paper, we introduce a new hybrid sharing technique providing checkpoint based explicit memory management. It consists in co-scheduling parallel applications within a set, until the memory capacity of the node is reached, and using gang scheduling related techniques to switch from one set to another one. We compare experimentally the merits of the three solutions: Co, Gang and Hybrid Scheduling, in the context of out-of-core computing, which is likely to occur in the Grid context, where many users share the same resources. The experiments show that the hybrid solution is as efficient as the co-scheduling technique when the physical memory is not exhausted, and is more efficient than gang scheduling and co-scheduling when physical memory is exhausted.

I. Introduction

Two of the most fundamental principles of Grid are 1) the capacity to establish virtual organizations spanning over the several administration domains in order to extend the number of resources accessible by users and 2) the coordination of these resources in order to cooperate in the resolution of user problems. From these two principles eventually arises the need for efficient and fair resource sharing mechanisms.

The first principle inevitably tends to increase the pressure on the system mechanisms implementing the resource sharing between users. In the general situation, the resources are belonging to institutions and are already used by some of their members. Thus, building virtual organizations on top of already used resources increases the number of potential users for these resources, leading to an increased requirement to share these resources fairly among the users. In large system within HPC centers, queues of jobs are already quite long. It is not uncommon to wait days before having large jobs done. If nothing is done in Grids associating tens of such sites, the waiting time would certainly evolves from days to weeks which in some circumstances will not be acceptable for users. An other fairness issue concerns the capacity to establish several level of priority between parallel executions on the Grid. High priority execution should be able to preempt the resource of a low priority parallel application under execution.

The second principle, as examined in the light of the first one, implies the use of efficient coordination mechanisms ensuring A) parallel application performance close to the one obtained in a dedicated system and B) limiting strictly the waste of computing resources by providing rapid parallel applications context switch. In this paper, we restrict our investigation domain to users running MPI applications on clusters shared within a Grid virtual organization. We focus on the following scenario: users submit their MPI jobs to a meta-scheduler (from a portal) which schedules them on dynamically selected clusters. Any MPI execution may span over several clusters or stay within a single cluster. In the first case, an efficient coordination mechanism should schedule simultaneously all the components involved in each MPI execution. The efficiency in fulfilling criteria A) and B) depends on the synchronization mechanism coordinating the scheduling of the MPI subparts over all the clusters involved in the execution and on the speed of context switch between the MPI executions on each cluster. In the second case, clusters are not synchronized and the efficiency of the Grid only depends on the capability of each cluster management system to meet the criteria A) and B).

Fairness and performance are in principle contradictory. Reaching top performance on parallel execution involves dedicated usage of the cluster resources. The less the operating system interrupts the application execution, the highest is the performance. Usually, fairness relies on context switch mechanisms enabling the resource sharing between users. Context switching adds an overhead on the total application execution time. The more context switches are experienced during an execution, the more the application is slowed down.

Sharing fairly the cluster resources between multiple users may be examined in two cases: 1) when the concurrent executions of all users fit in the memory of the cluster nodes and 2) conversely when disk storage should be used in addition to the memory in the cluster nodes to store all concurrent executions. We will call these two cases respectively in-core and out-of-core context switch.

Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)
1550-5510/04 $ 20.00 IEEE

The two principles lead to three main consequences: 1) a fast mechanism for switching the context of MPI execution on a cluster is the corner stone to meet efficiency criteria, 2) because fairness and performance are contradictory objectives, we should consider a metric representing a tradeoff between them. For shake of simplicity, in this paper, we will consider application with the same execution time and the following ratio as the tradeoff metric : the execution time of a set of parallel applications over the standard deviation of their individual execution times and 3) in a Grid with lot of users, it is likely that out-of-core context switch will be the general case. Thus in this paper, we will essentially focus on this context.

In this paper we study several MPI application context switching techniques, trying to discover which one has the lowest impact on application performance. We will demonstrate that the best technique for out-of-core context switching is a hybrid (two level) one mixing checkpoint based context switching of sets of MPI executions and uncoordinated scheduling (co-scheduling) of MPI executions within each set.

The second section presents the related works. Section III presents the general framework used to compare the different scheduling techniques. Section IV presents the different scheduling approaches compared in this paper. Section V presents the experimental results and section VI concludes and sums up what we learned from the experiences.

II. Related work

There are several main techniques to implement parallel application context switch in practice. One of the most used one is batch scheduling, queuing the jobs submitted by the different users. In the general case, after being elected for execution, parallel jobs are scheduled subsequently (one after the other). Thus, only one parallel execution runs on the cluster at a given time. Example of existing implementation of batch schedulers are PBS[1], LSF, Condor, etc. The main drawback of the batch scheduling approach is its lack of fairness between users submitting heterogeneous jobs. An application that needs a large number of nodes but for a short period may have to wait until all longer jobs running on fewer number of nodes terminate before being allowed to run.

This paper focuses on another family of approaches which lets the operating system schedules the processes of several parallel executions launched concurrently, according to their priorities. These techniques are called gang scheduling and co-scheduling, depending on the scheduling coordination of parallel execution processes. There is no coordination in co-scheduling. In gang scheduling, processes of a given application are scheduled simultaneously, requiring some synchronization mechanism. In these techniques all concurrent parallel applications reside in the cluster memory until their completion, generally leading to a huge usage of the virtual memory system.

[2], [3] has proposed one of the first implementation of gang scheduling, called SCore. SCore targets clusters and is based on a *Network preemption* procedure relying on the PM communication library [4]. The gang scheduling itself is performed using UNIX signal mechanism. When an application have to be unscheduled, all it's processes on all nodes receive a SIGSTOP signal. Thus when another application is scheduled by the reception of the SIGCONT signal, it gets exclusive usage of computational power and network resources. However, the memory is shared between running and stopped applications. Memory sharing is resolved by the virtual memory mechanism of the operating system, as inactive applications may be transfered in swap memory. The gang scheduling strategy we present in this paper explicitly stores and reloads stopped processes, limiting the memory sharing between applications. Our approach also does not rely on operating system swapping mechanism. Another difference compared to SCore is the network flush algorithm. SCore does not use the Chandy & Lamport algorithm to flush the network, but a three phases synchronization algorithm using the PM flow-control protocol. The Chandy & Lamport algorithm we implemented uses only one synchronization phase. In checkpoint based scheduling typically, only one execution resides in memory at a given time. Note that uploading and downloading executions to and from the memory involves disk operations which can add a significant overhead.

An example of gang scheduling evaluation on LLNL Cray T3D can be found in [5].

[6] evaluates different scheduling policies using the LSF batch scheduler. All policies are refinements of gang scheduling techniques allowing each application to run solely on the required processors. Three classes of applications are considered: short (5 minutes termination expected time), medium (60 minutes) and long running (no limit). The paper studies different policies when an application with a shorter termination expected time is queued. They demonstrate that preemption, implemented with checkpointing techniques, is crucial to obtain good response time. In this paper we extend the notion of hybridness, mixing gang scheduling with co-scheduling. The resulting approach could be adapted to work within a batch scheduler. The experimental studies concerning this issue will be presented in a future paper.

Co-scheduling, consists in launching all applications on the system resources and letting each node operating system scheduling the different jobs. The lack of coordination for the simultaneous execution of all nodes job of an application is a major drawback for synchronous operations, but this approach allows better overlap communication of a job by computation of another one (we assume that the communications are buffered independently of the process scheduling, like in kernel level protocol stacks). As the co-scheduling relies on the operating system memory scheduling, running out-of-core applications leads to high overhead due to the system swap policy. [7] introduces different paging techniques to reduce the number of page faults.

Some studies have compared and mixed gang scheduling and co-scheduling. In [8] the authors analyze experimentally that co-scheduling outperforms gang scheduling for clusters and in-core running applications. Our study pushes this result

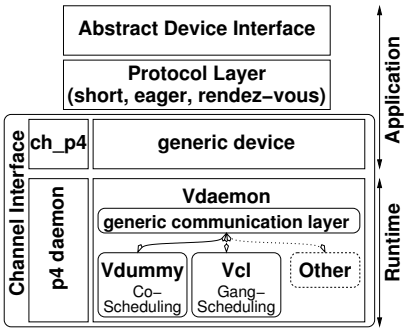


Fig. 1. Architecture of MPICH-V compared to architecture of MPICH-P4

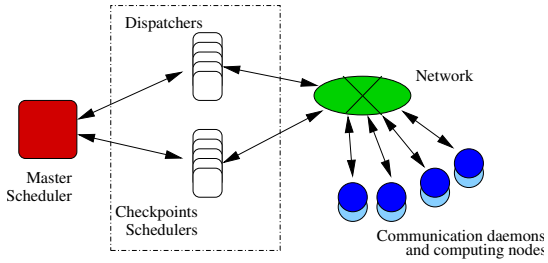


Fig. 2. Typical deployments of MPICH-V with many applications

in the case of out of core running applications, a major concern for Grid systems as the number of users and tasks expands. [9], [10], [11] consider determined classes of application based on their communication and I/O characteristics. Thus they improve gang scheduling of such known applications by co-scheduling applications of different classes. In this paper, we demonstrate that mixing the two techniques provides the best performance in the out-of-core case, even if applications are identical.

III. Common framework

In this paper, we focus on comparing and mixing two scheduling methods for MPI application time sharing. The first one is based on the ability to stop and restart a set of MPI applications so that memory used by these applications is available to another set of applications. At the end of a time slice, a set of applications is dumped on disk, and another set of applications is loaded from a previous checkpoint and run during the next time slice. The second one is based on running all MPI applications simultaneously on the same set of nodes. This method relies on the operating system scheduler to perform time sharing and is called co-scheduling.

The MPICH-V framework offers both stop and restart capable and standard MPI implementations. It's main focus is comparison of different kind of fault tolerant protocols for MPI. Deriving from this main purpose, we developed two non fault tolerant protocols. One including distributed checkpoint facility based on the Chandy&Lamport algorithm (Vcl [12], [13]), and another one implementing basic MPI communication, without checkpoint capabilities (Vdummy). Using such an implementation is mandatory to perform a fair comparison: as the two implementations share the same framework, any performance difference is related to the scheduling itself, and not to implementation optimizations. Moreover, we compared the MPICH-V framework to the reference implementation MPICH-P4 in [13]. Figure 1 compares the architectures, while figure 3 recalls these performances comparison between the MPICH-V framework and the reference implementation, and validates the checkpoint enabled MPICH-Vcl version performance compared to standard MPICH-Vdummy version.

MPICH-V is based on the MPICH library [14], which builds a full MPI library from a channel. A channel implements the basic communication routines for a specific hardware or for new communication protocols. MPICH-V consists in a set of runtime components and a channel (ch_v) for the MPICH library.

The different protocols are implemented in the MPICH-V framework at the same level of the software hierarchy, between a MPI high level protocol management layer (managing global operations, point to point protocols, etc.) and the low level network transport layer. Among the other benefits, this allows to keep unmodified the MPICH implementation of point to point and global operations, as well as complex concepts such as topologies and communication contexts. A potential drawback of this approach might be the necessity to implement a specific driver for all types of Network Interface (NIC). However, several NIC vendors provide low level, high performance (zero copy) generic socket interfaces such as Socket-GM for Myrinet, SCI-Socket for SCI and IPoIB for Infiniband. MPICH-V protocols typically seat on top of these low level drivers. So this is one of the most relevant layer for implementing new MPI capabilities if criteria such as design simplicity, high performance, heterogeneous network migration and portability are to be considered.

MPICH-V provides all the components necessary to stop and restart MPI applications. Some of them have been slightly modified to focus on the scheduling of MPI applications, while some have been added, specifically to manage sets of applications.

A. Dispatcher

The dispatcher of the MPICH-V environment has two main purposes: 1) to launch the whole runtime environment (encompassing the computing nodes and the auxiliary "special" nodes) on the pool of machines used for the execution, and 2) to monitor this execution, by detecting node disconnection and then stop the execution.

The Dispatcher is in charge of a single MPI application. If more than one application is running at a time on a cluster, each is controlled by it's own Dispatcher.

B. Driver

The driver is the part of the MPICH-V framework linked with the MPI application. It implements the Channel Interface of MPICH. Our implementation only provides synchronous functions (bsend, breceive, probe, initialize and finalize), as the asynchronism is delayed to another component of the architecture.

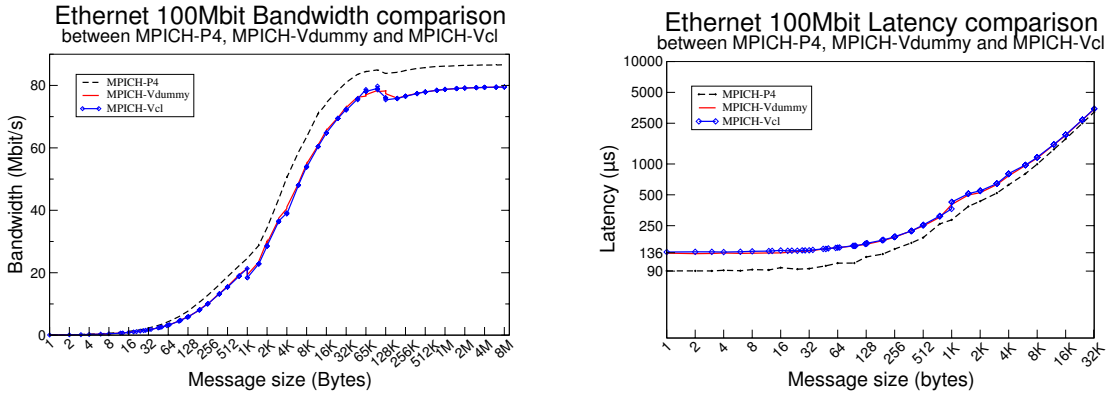


Fig. 3. Bandwidth and latency comparison between MPICH-P4, MPICH-Vdummy, and MPICH-Vcl on Fast-Ethernet network.

C. Communication daemon

The core of the communication daemon is a select loop: it manages one socket for every computing node and one socket for every specific components. Every send or receive operation is asynchronous. Thus, a communication is not blocked by another slower one. At the contrary, the communication across the Inter Process Communication mechanism to the MPI process is synchronous and its granularity is the whole protocol message. The communication daemon is in charge of all distributed checkpoint mechanisms.

The checkpoint of the daemon uses an explicit serialization of its data and when a checkpoint is requested, some messages have to be logged on the receiver (considered as the in-transit messages of the Chandy-Lamport algorithm).

a) The generic Communication daemon: Daemons implements a generic communication layer to provide all the communication routines between the different kind of components involved in the MPICH-V architecture, independently of the protocols. Checkpoint enabled protocols are designed through the implementation of a set of hooks called in relevant routines of the generic layer and some specific components (figure 1).

The collection of all these functions is defined through a fault tolerance API and each protocol implements this API. In order to reduce the number of system calls, communications are packed using *iovec* related techniques by the generic communication layer. The different communication channels are multiplexed using a single thread and the *select()* system call. This common implementation of communications eases the implementation of protocols and allows a fair comparison between them.

D. Checkpoint scheduler

The checkpoint scheduler requests computation processes to checkpoint according to a given policy. The policy is protocol dependent. In this article, the checkpoint scheduler uses a coordinated checkpoint policy driven by the Master Dispatcher, intended to checkpoint an application before it is stopped at the end of a time slice.

E. Master Scheduler

The Master Scheduler is a new component introduced in the MPICH-V architecture in order to target Grid scheduling. The master scheduler coordinates the scheduling of all the applications on the system. Given a list of applications to schedule, it first launches the dispatcher and checkpoint scheduler component of each application. The application deployment itself is performed by the Dispatcher of this application (figure 2).

The number n of job to run simultaneously is given as parameter of the master scheduler. The Master Scheduler associates a time slice to each application. When a scheduled application has been running long enough to expire its time slice, the Master Scheduler requests the Dispatcher and the Checkpoint Scheduler to stop this application. When it is stopped, the Master Scheduler requests another Dispatcher to restart the associated application, and the time slice for this application begins.

The Master Dispatcher implements various policies for stopping/restarting applications. It is possible to Co-schedule k applications of a set of n. To perform set context switch, three checkpoint/restart overlap policies are implemented. These policies are detailed in section IV-B.2.

IV. Three techniques of scheduling

We study three different kinds of scheduling for sharing multiple MPI application on a single set of nodes. Two of them, co-scheduling and gang scheduling, were compared in the context of clusters and using applications with low memory usage in [8]. These two techniques were not studied in the context of a large memory usage, leading to out-of-core computation when many applications runs simultaneously. We propose a new hybrid method based on the mix of the co-scheduling and the gang scheduling. The main idea is to limit the number of concurrent co-scheduled applications using gang scheduling global time slice and checkpoint related techniques, so that physical memory would not be exhausted by co-scheduling all applications.

A. Literature techniques of time-sharing

1) Co-scheduling: The first one, called co-scheduling consists in an uncoordinated approach. The processes of each

application are launched simultaneously on all nodes. On each node, the local operating system scheduler is in charge of sharing resources between the processes of different applications. Thus processes of an application may not be scheduled at the same time on all nodes, which could impact performances of applications using a tightly synchronized communication scheme. Moreover, the frequent context switches between processes may introduce many cache faults. Nonetheless, This technique requires no specific implementation, and computational and network resources may be better used, like in the multi-threaded programming scheme.

To perform comparison of this technique to others, we used the MPICH-V environment. The MPICH-Vdummy implementation does not include any checkpoint or time-sharing ability, and is a good candidate to perform a fair comparison with the other scheduling techniques we study. The Master Scheduler uses a policy which just spawn all applications on the nodes simultaneously, and then waits for termination, relying on local node's operating system to schedule processes of all applications.

2) Gang scheduling: The second approach is called gang scheduling. It consists in synchronizing all local scheduler so that all processes of a single distributed application are scheduled simultaneously, while all other applications are stopped and sleeping. All resources are thus employed on the execution of a single application, avoiding the wait for messages from a process not scheduled on another node. However, no multi-thread effect is possible as only one application is running at a time. As discussed in [8], on many applications, the co-scheduling technique outperforms gang scheduling, as applications do not perfectly overlap communications with computation.

Gang scheduling is implemented in the Master Scheduler of the MPICH-V framework.

B. New hybrid technique of time-sharing for high memory requirement

In this method, we propose to use co-scheduling for in-core computation, as it has been proven to be more efficient than gang scheduling. When out-of-core computation would appear using co-scheduling, we use a gang scheduling related technique to enforce that only a subset of the applications is running on the nodes. As the overhead induced by the applications switch is related to time to store process memory on local disk, it is much higher than node operating system context switch overhead. As a consequence, time slices have to be much longer. Thus, gang scheduling is mandatory to reach good performance for any communicating application. Waiting for a message during the full time slice of an application would lead to very poor network performance.

The figure 4 presents a typical deployment of the hybrid architecture. In this example we suppose that physical memory size allows to run two simultaneous co-scheduled applications without inducing out-of-core computation.

The Master Scheduler controls Dispatchers and Checkpoint Schedulers of each application, enforcing some applications to

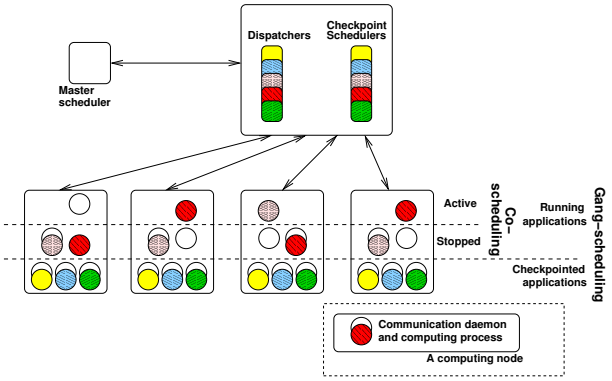


Fig. 4. Typical deployment of hybrid scheduling

be stopped and swapped out of memory, while some others are restarted. We explain in section IV-B.2 different methods to swap out of memory an application. For the set of applications that are running, their processes are co-scheduled by each node operating system.

1) Network management of application switching: To stop an application, network status have to be saved. We use the Chandy&Lamport algorithm to flush network before the application is stopped. The Checkpoint Scheduler and the communication daemons are dedicated to save the network state. The checkpoint scheduler requests every communication daemon to take a global snapshot by sending a tag in each communication channel. On the reception of this tag, a daemon stops the computing process, and then sends the tag in every communication channel. When the checkpoint scheduler have received a tag from every communication daemon, the network flush is achieved.

2) Memory management of application switching:

a) SIGSTOP/SIGCONT policy, system memory management: In previous implementations of gang scheduling, each process of an application are stopped by the SIGSTOP UNIX signal. In the case of large memory consumption, memory sharing is managed by operating system swap policy. As running applications were swapped on disk during the previous time slice, the memory pages are reloaded on demand during the execution (thus swapping out some pages used by stopped applications). The number of page faults has a major impact on overall performance. The overhead using this technique is very unpredictable as it relies on operating system swapping policy. Moreover, relying on SIGSTOP/SIGCONT mechanism would introduce perturbations in our page fault measurements, as it is difficult to differentiate out-of-core computation and context switch induced page faults. Thus, we prefer to use a checkpoint based technique, which overhead is well bounded.

b) Checkpoint policies, explicit memory management: In our implementation, we use checkpoint/restart to explicitly manage memory swapping of applications memory. When a computing process is requested to be stopped, it performs a checkpoint to local disk, thus freeing all memory it uses. Complete memory of process to be scheduled in the next time slice is reloaded from checkpoint, thus no page fault occurs during the time slice. Incremental checkpoint related

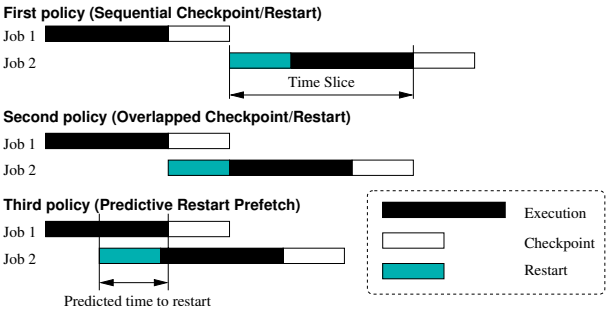
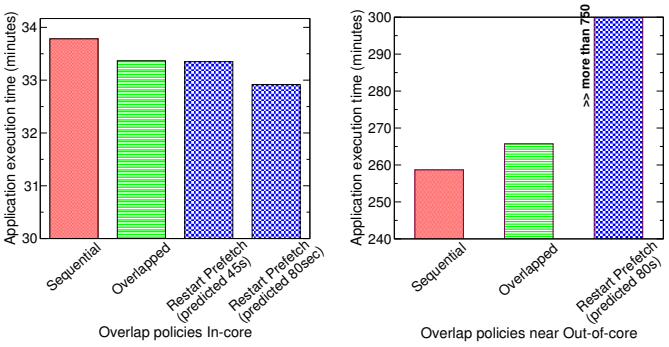


Fig. 5. Policy for checkpointing a job and restarting the next scheduled one on the same node



(a) 2 Hybrid scheduled BT.C.25, 1 running at a time (in-core).

(b) 10 Hybrid scheduled BT.C.25, 5 running at a time (near out-of-core).

Fig. 6. Checkpoint / Restart application context switch policies comparison, using NAS benchmark BT class C on 25 nodes on Ethernet performance criteria.

techniques may be used to improve checkpoint performances. This memory management is equivalent to aggressive paging out technique described in [7].

Many policies may be used to overlap checkpoint, restart, and computations. Figure 5 presents the three techniques implemented in the comparison framework. The first method (called sequential checkpoint/restart) does not overlap checkpoint and restart. It avoids to load the two applications simultaneously in memory at the expense of serializing checkpoint, restart and computation during the context switch.

The second technique (called overlapped checkpoint/restart) overlaps checkpoint and restart, trying to reduce context switch cost. It requires more memory as one application is reloaded before the previous is fully flushed out of memory, and induces simultaneous disk accesses.

The third technique (called predictive restart prefetch) tries to prefetch restart during the end of the time slice of the previous application, so that restart is overlapped by computation of the application to be stopped, and checkpoint is overlapped by computation of the next application. It has to rely on an oracle, predicting time to restart, and induces simultaneous memory usage during the end of the time slice of the application to be stopped.

These three techniques are compared in section V-B.

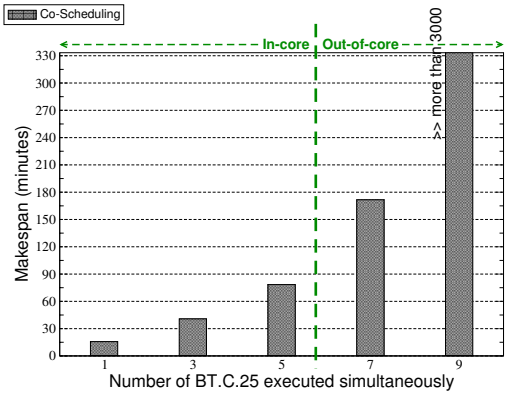


Fig. 7. Makespan for n simultaneous BT class C 25 nodes using Co-scheduling.

V. Performance Evaluation

A. Experimental conditions

We present a set of experiments in order to evaluate the different components of the system.

Experiments are run on a 32-nodes cluster. Each node is equipped with an Athlon XP 2800+ processor, running at 2GHz, 1GB of main memory (DDR SDRAM), and a 70GB IDE ATA100 hard drive and a 100Mbit/s Ethernet Network Interface Card. Swap space is set to 10GB. All nodes are connected by a single Fast Ethernet Switch.

All these nodes use Linux 2.4.21 as operating system. The tests and benchmarks are compiled with GCC 2.95-5 (with flag -O3) and the PGI Fortran77 compilers. All tests are run in dedicated mode. Each measurement is repeated 5 times and we present a mean of them.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. We use the NetPIPE [15] utility to measure bandwidth and latency of the MPICH-V framework. This is a ping-pong test for several message sizes and small perturbations around these sizes. The second set of experiments is the CG and BT kernel and application of the NAS Parallel Benchmark suite [16], written by the NASA NAS research center to test high performance parallel computers. These two benchmarks cover a wide spectrum of applications and communication patterns. Each process of a BT benchmark uses 175MB of memory for class C on 25 nodes, 135MB for class B on 9 nodes, and each process of CG class C on 8 nodes uses 157MB. A node has 1GB of memory (900 MB of user space memory). Thus up to five simultaneous applications fit in physical memory whatever the application we use. Moreover from 7 simultaneous applications, swap is always used.

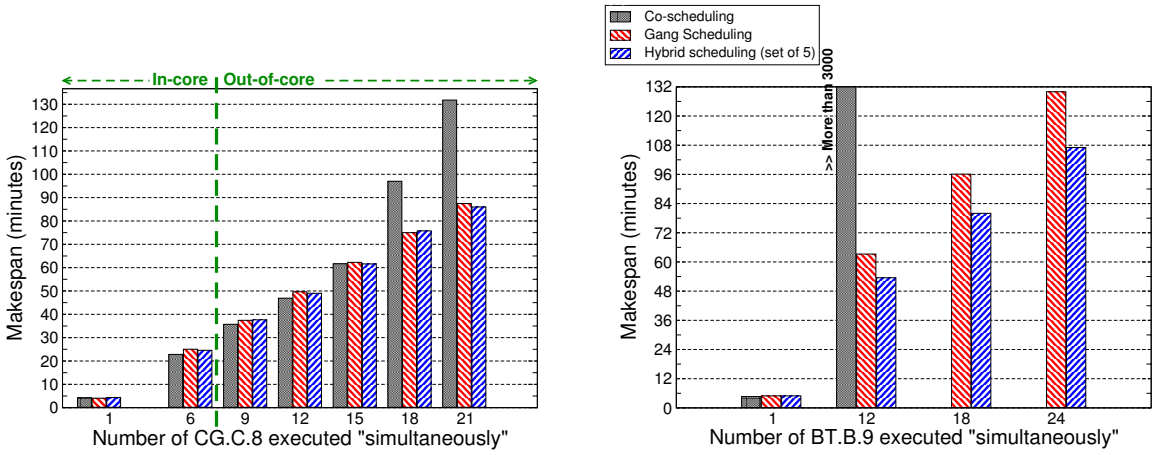
B. Checkpoint/restart overlap scheduling policy

Figure 6 presents the performance of NAS benchmark BT class C on 25 nodes for the three application context switch policies of Checkpoint/Restart presented in section IV-B.2.b. On the first figure, two concurrent applications are gang scheduled. On the second figure, 5 applications of 10 are run simultaneously, thus filling physical memory.

When physical memory is not exhausted by computing applications, as expected, the more overlap is reached, the bet-

Number of applications	Average number of major page faults for all nodes of each application, per minutes									Average page faults per minutes, All applications	Standard deviation
	app 0	app 1	app 2	app 3	app 4	app 5	app 6	app 7	app 8		
7	***8.5***	***239.5***	***264.25***	951	1145.25	***1245.5***	1074			704	**474.9**
9	484.58	405.94	564.78	524.4	510.66	577.26	506.94	481.84	509.4	507.4	**47.1**

Fig. 8. Page faults statistics for simultaneous BT class C on 25 nodes



(a) CG class C 8 nodes (b) BT Class B 9 nodes

Fig. 9. Makespan for n simultaneous applications using Co-scheduling, Gang scheduling or Hybrid-scheduling.

ter performance is. Thus, the Overlapped Checkpoint/Restart method performs better than Sequential Checkpoint/Restart, and the Predictive Restart Prefetch method performs better than the two others. Moreover, it is better to use a greedy prefetch than an exact one, as the predicted checkpoint time may vary around the mean value.

When running multiple applications up to exhaust physical memory, the overlapping strategies do not perform as well. The finishing application stays in memory during its checkpoint, thus inducing memory swapping for computing restarted applications. Moreover, the restart prefetch strategy suffers from a dramatic overhead related to the very intensive memory usage induced by the simultaneous execution of applications at the end of their time slice and applications beginning a new time slice. Concurrent applications swapping on disk increase the checkpoint time, while checkpoint and restart accesses on disk decrease swapping performance. Thus, the more the disk is accessed simultaneously, the more the overall performance is decreased. As it sequentializes both disk access and memory usage, the sequential policy performs better when physical memory is near to be filled.

C. Scheduling techniques performance comparison

Figure 7 presents the computation time of n simultaneous BT class C on 25 nodes benchmark. When the memory used by the concurrent applications fits in-core, the computation throughput increases slightly with the number of applications. For 7 simultaneous applications, the overall throughput is decreasing, and for 9 simultaneous applications, the makespan is twenty times the sequential batch scheduling makespan. The figure 8 explains this result. It presents the average number of major page faults per minute for 7 and 9 simultaneous applications using Co-scheduling. On one hand, for 7 Co-scheduled applications, only a subset of the applications hits the swap: the first application does not suffer from any page fault (8.5 page faults per minutes), while another suffers from more than 1245 page faults per minutes. On the other hand, for 9 Co-scheduled applications, all applications hit the swap equally (Standard deviation is less than 47.1 for an average page faults per minute of 507.4). This outlines the lack of fairness of the virtual memory management used in the Linux 2.4.21 kernel when only a small amount of swap is used. This algorithm reaches good performance by scheduling more the in-core applications, at the cost of serializing executions. When memory occupation leads the size of the page cache table to reach its lower bound *pages_table_low*, the virtual memory manager applies a first "gentle" policy, which is the case for 7 simultaneous applications. For 9 simultaneous applications and above, the upper bound *pages_table_high* is reached, setting the virtual memory manager in an "aggressive" (but fair) swapping policy. In this case, all applications have fair access to physical memory, but performance suffers from a dramatic decrease. Even if these bounds may be tuned, at some point, the kernel has to switch to the aggressive policy, in order to ensure availability and fairness.

Figure 9 presents the computation time of n simultaneous NAS benchmarks, using co-scheduling, gang scheduling or hybrid scheduling. Hybrid scheduling relies on operating system scheduler for executing a set of 5 applications simultaneously, in order to occupy all the physical memory without inducing out-of-core computation. The sequential Checkpoint/Restart policy is used to perform set of applications context switch. Time slices are 900 seconds.

For the CG benchmark (figure 9(a)), the co-scheduling

method reaches good performance up to use about 3 times the physical memory (2800MB). Comparing to the BT benchmark results (figure 7), this outlines that the point of inefficiency of co-scheduling is tightly related to the pattern of memory accesses. The application uses often the same pages, reducing the number of page faults. However, for these two applications, the huge impact on co-scheduling of the kernel aggressive swapping policy is observed. When co-scheduling is outperformed, gang and hybrid scheduling techniques reach the same performance. This outlines that there is almost no benefit of co-scheduling a subset of applications for this benchmark. In CG, communication prevails, thus the bandwidth is divided between co-scheduled applications.

The figure 9(b) focuses on performance comparison between gang scheduling and hybrid scheduling when co-scheduling is outperformed due to out-of-core computation for the BT benchmark. For each time slice, gang scheduling and hybrid scheduling have to checkpoint and restart one application. During that time slice, no swap effect occurs, as the number of applications running simultaneously is either one for gang scheduling or fits in-core for hybrid scheduling. Obviously, the performance is exactly the performance of pure gang scheduling compared to pure in-core co-scheduling. Unlike the CG benchmark, computation prevails in the BT benchmark. For this kind of applications, hybrid scheduling performs 20% better than gang scheduling.

For every benchmark, overall throughput is equal or better to sequential scheduling when using hybrid scheduling.

VI. Conclusion

In this paper, we compare several scheduling techniques for the Grid. Some well known as gang scheduling (which schedules a whole single application on all nodes at a given time) and co-scheduling (which schedules all applications simultaneously on all nodes). We propose a new scheduling approach, based on the two previous, that we call hybrid scheduling.

Hybrid scheduling consists in splitting the set of applications to schedule in subsets, co-scheduling the applications of a same subset and gang scheduling the different subsets. The main decisive factor is the amount of physical memory used by all the processes of a same subset. The technique uses user-mode checkpoints to stop and restart gang scheduled applications. Using three different policies, we studied experimentally the merits of overlapping or not the checkpoint and restart during gang scheduling context switch.

We conducted a set of experiments using the NAS parallel benchmarks. We show that for out-of-core computation, co-scheduling behaves accordingly to the swap policy of the operating system, and is eventually hit by very poor performance. We show that according to the application computation communication ratio, hybrid scheduling compares favorably or equally to gang scheduling. Comparing to sequential batch scheduling, hybrid scheduling reaches slightly better or equal throughput, and offers a better fairness.

We plan to improve the master dispatcher, in order to dynamically adapt the number of applications running concurrently to the memory occupation of nodes. Another issue is the gang scheduler context switch efficiency, so we plan to compare user-space checkpoint techniques to optimized kernel swap algorithm in the context of overflowed physical memory. Finally, we plan to integrate our hybrid scheduling system into a meta batch scheduling system for the grid, to compare the fairness, reactivity and performance of such a system with classical batch scheduling.

References

[1] R. L. Henderson, "Job scheduling under the portable batch system," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pp. 279 – 294.

[2] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "Implementation of gang-scheduling on workstation cluster," in *Proceeding of IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer-Verlag, April 1996, pp. 126–139.

[3] A. Hori, H. Tezuka, and Y. Ishikawa, "Overhead analysis of preemptive gang scheduling," *Lecture Notes in Computer Science*, vol. 1459, pp. 217–230, April 1998.

[4] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato, "PM: An operating system coordinated high performance communication library," in *Proceedings of the international conference and Exhibition on high-Performance Computing and Networking*. Springer-Verlag, April 1997, pp. 708–717.

[5] D. G. Feitelson and M. A. Jette, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., vol. 1291. Springer Verlag, 1997, pp. 238–261.

[6] E. W. Parsons and K. C. Sevcik, "Implementing multiprocessor scheduling disciplines," in *Proceeding of IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processin*, D. G. Feitelson and L. Rudolph, Eds. Springer Verlag, April 1997, pp. 166–192.

[7] K. D. Ryu, N. Pachapurkar, and L. L. Fong, "Adaptive memory paging for efficient gang scheduling of parallel applications," in *18th International Parallel and Distributed Processing Symposium IPDPS'04*, April 2004.

[8] P. Strazdins and J. Uhlmann, "Local scheduling out-performs gang scheduling on a beowulf cluster," Department of Computer Science, Australian National University, Tech. Rep., January 2004.

[9] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, June 2003, pp. 581–592.

[10] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "Implications of I/O for gang scheduled workloads," *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pp. 215–237, 1997.

[11] F. A. B. da Silva and I. D. Scherson, "Improving parallel job scheduling using runtime measurements," *Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 18–38, May 2000.

[12] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, December 2003.

[13] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *to appear in IEEE International Conference on Cluster Computing (Cluster 2004)*, 2004.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[15] Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *IASTED International Conference on Intelligent Information Management and Systems.*, June 1996.

[16] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Report NAS-95-020, 1995.