HYBRID PREEMPTIVE SCHEDULING OF MESSAGE PASSING INTERFACE APPLICATIONS ON GRIDS

Aurélien Bouteiller
Hinde-Lilia Bouziane
Thomas Herault
Pierre Lemarinier
Franck Cappello

INRIA/LRI UNIVERSITÉ PARIS-SUD, ORSAY, FRANCE
(BOUTEILL@LRI.FR)

Abstract

Time sharing between cluster resources in a 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 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 outperform 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-scheduling, gang scheduling, 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. Additionally, we address the problem of heterogeneous applications by comparing hybrid scheduling to an optimized version relying on paired scheduling. The experiments show that the hybrid solution is as efficient as the co-scheduling technique when the physical memory is not exhausted, can benefit from the paired scheduling optimization technique when applications are heterogeneous, and is more efficient than gang scheduling and co-scheduling when physical memory is exhausted.

Key words: Scheduling, performance, message passing interface, time sharing, grid, gang scheduling, co-scheduling

The International Journal of High Performance Computing Applications,
Volume 20, No. 1, Spring 2006, pp. 77–90
DOI: 10.1177/1094342006062526

Figures 7–11 appear in color online: http://hpc.sagepub.com

1 Introduction

Two of the most fundamental principles of grids are (1) the capacity to establish virtual organizations spanning over 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 solving 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 belong 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 systems within high performance computing 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 evolve from days to weeks, which in some circumstances will not be acceptable for users. Another fairness issue concerns the capacity to establish several level of priority among parallel jobs on the grid. High-priority jobs 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, implies the use of efficient coordination mechanisms ensuring (a) parallel application performance close to that 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 message passing interface (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 remain 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 fewer times the operating system interrupts the application execution, the higher the performance. Usually, fairness relies on context switch mechanisms enabling 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 call these two cases in-core and out-of-core context switching, respectively.

The two principles lead to three main consequences. (1) A fast mechanism for switching the context of MPI execution on a cluster is the cornerstone to meet efficiency criteria. (2) Because fairness and performance are contradictory objectives, we should consider a metric representing a trade-off between them. For the sake of simplicity, in this paper, we consider applications with the same execution time. (3) In a grid with many users, it is likely that out-of-core context switching will be the general case. Thus, in this paper, we 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 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. We also demonstrate that hybrid scheduling can benefit from the optimizations to gang and co-scheduling presented in previous works (paired scheduling).

In Section 2 we present the related works. In Section 3 we present the different scheduling approaches compared in this paper. In Section 4 we present the general framework used to compare the different scheduling techniques. In Section 5 we present the experimental results and in Section 6 we conclude and sum up what we have learned from the experiences.

2 Related Work

There are several main techniques to implement parallel application context switching in practice. One of the most used is batch scheduling, i.e. queuing the jobs submitted by the different users. In the general case, after being elected for execution, parallel jobs are scheduled sequentially (one after the other). Thus, only one parallel execution runs on the cluster at a given time. Examples of existing implementations of batch schedulers are PBS (Henderson 1995), 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 a fewer number of nodes terminate before being allowed to run.

In this paper we focus on another family of approaches which lets the operating system schedule 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 demand on the virtual memory system.

Hori et al. (1996, 1998) have proposed one of the first implementations of gang scheduling, called SCore. SCore targets clusters and is based on a network preemption procedure relying on the PM communication library (Tezuka et al. 1997). The gang scheduling itself is performed using a UNIX signal mechanism. When an application has to be unscheduled, all its processes on all nodes receive a SIGSTOP signal. Thus, when another application is scheduled by the reception of the SIGCONT signal, it has 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 transferred in swap memory. The gang scheduling strategy we present in this paper explicitly stores and reloads stopped processes and does not rely on operating system swapping mechanism. In our checkpoint-based scheduling, typically only one execution resides in memory at a given time, thus limiting the memory sharing between applications. 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-phase synchronization algorithm using the PM flow-control protocol. The Chandy–Lamport algorithm we implemented uses only one synchronization phase. 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's Cray T3D can be found in Feitelson and Jette (1997).

Parsons and Sevcik (1997) evaluate 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 min termination expected time), medium (60 min) and long running (no limit). They study 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 involves launching all applications on the system resources and letting each node's operating system schedule the different jobs. The lack of coordination for the simultaneous execution of each node's part of an application is a major drawback for synchronous operations, but this approach allows a better overlapping of the communication of one job with the computation of another one (we assume that the communications are buffered independently of the process scheduling, as in kernel level protocol stacks). As co-scheduling relies on the operating system's memory scheduling, running out-of-core applications leads to high overhead due to the system swap policy. Ryu, Pachapurkar, and Fong (2004) introduce different paging techniques to reduce the number of page faults.

Some studies have compared and mixed gang scheduling and co-scheduling. Strazdins and Uhlmann (2004) demonstrate experimentally that co-scheduling outperforms gang scheduling for clusters and in-core running applications. Our study pushes this result in the case of out-of-core applications, a major concern for grid systems as the number of users and tasks expands. Wiseman and Feitelson (2003), Lee et al. (1997), and da Silva and Scherson (2000) consider specific 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 the applications are identical.

3 Three Techniques of Scheduling

We study three different kinds of scheduling for sharing multiple MPI applications on a single set of nodes. Two of these, co-scheduling and gang scheduling, were compared in the context of clusters and using applications with low memory usage in Strazdins and Uhlmann (2004). These two techniques were not studied in the context of a large memory usage, leading to out-of-core computation when many applications run simultaneously. We propose a new hybrid method based on the mix of co-scheduling and 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.

3.1 Co-Scheduling

The first kind of scheduling, called co-scheduling, consists of an uncoordinated approach. Figure 1 presents an

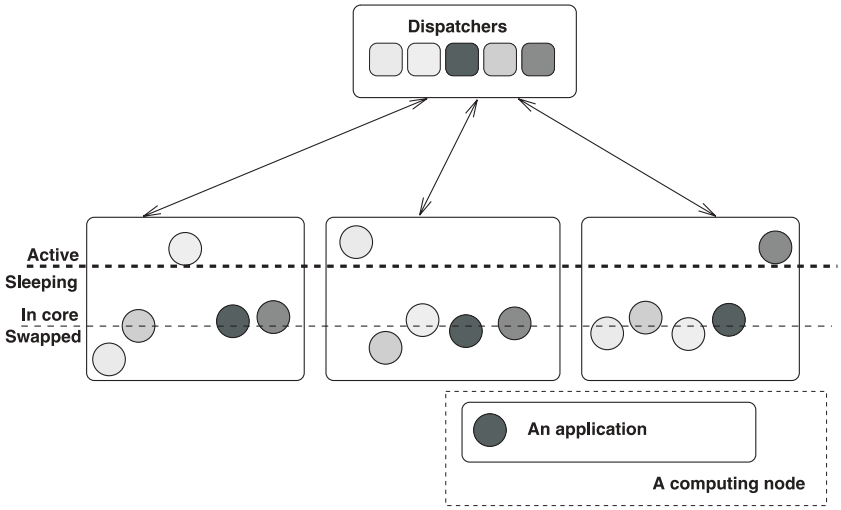

Fig. 1 Typical deployment of co-scheduling.

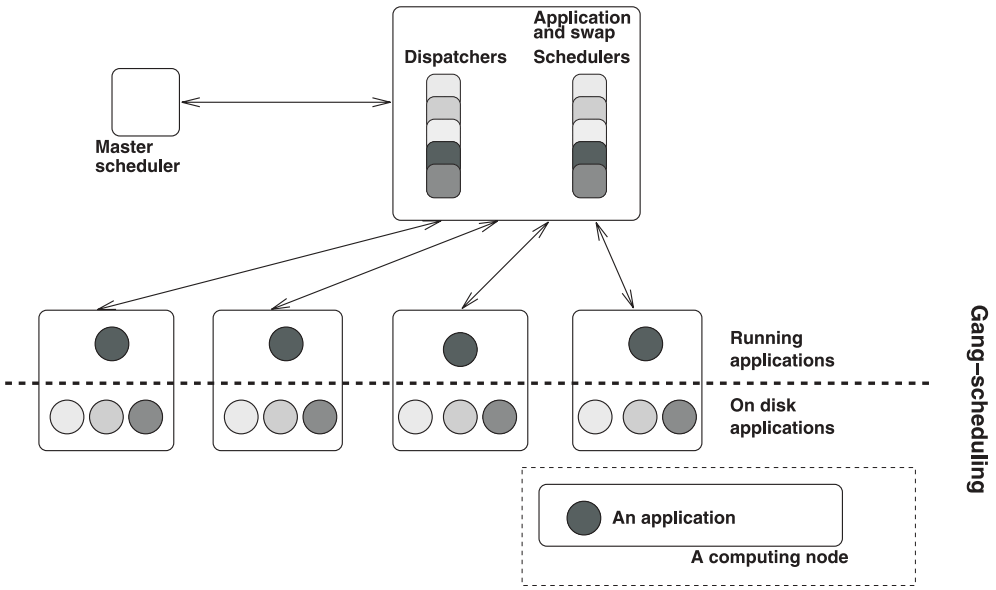

Fig. 2 Typical deployment of gang scheduling.

example of deployment of the co-scheduling architecture. In this example we suppose that three simultaneous applications fit in physical memory. 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 among 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, as in the multithreaded programming scheme.

3.2 Gang Scheduling

The second approach is called gang scheduling. Figure 2 presents a typical deployment of the gang scheduling architecture. It consists in synchronizing all the local schedulers 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 multithread effect is possible as only one application is running at a time. However, there is no cache fault, and all physical memory is dedicated to the currently scheduled application. As discussed in Strazdins and Uhlmann (2004), on many applications, the co-scheduling technique outperforms gang scheduling, as most applications do not perfectly overlap communications with computation.

Gang scheduling is implemented in the master scheduler of the MPICH-V framework.

3.3 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 proved 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 the node operating system context switch overhead. As a consequence, time slices have to be much longer. Thus, gang

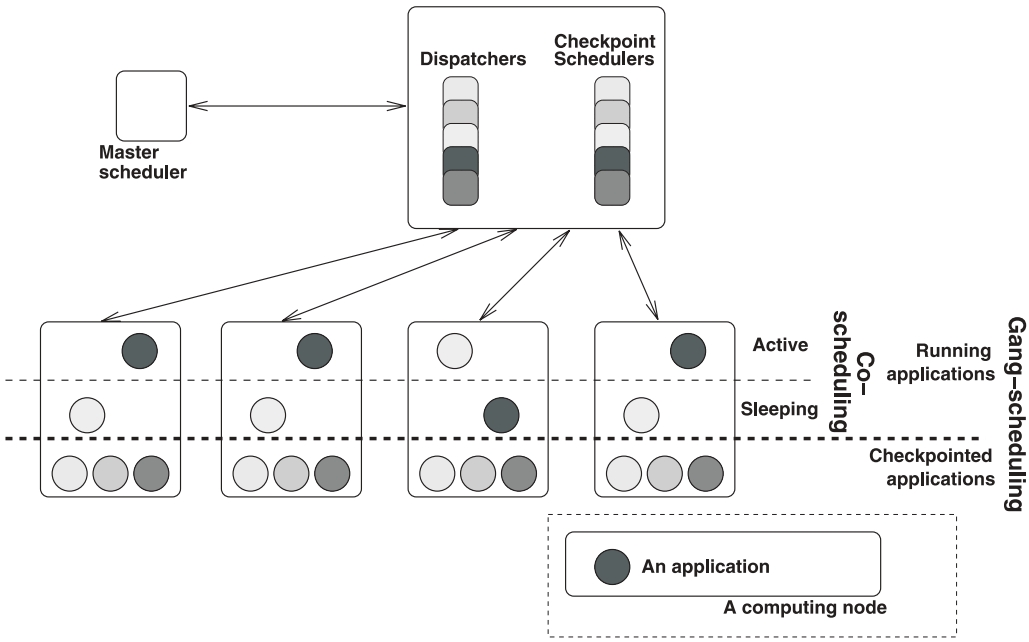

Fig. 3 Typical deployment of hybrid scheduling.

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.

Figure 3 presents a typical deployment of the hybrid architecture. In this example we suppose that physical memory size allows us to run two co-scheduled applications simultaneously without inducing out-of-core execution.

The master scheduler controls the dispatchers and checkpoint schedulers of each application, forcing some applications to be stopped and swapped out of memory, while some others are restarted. We explain in Section 3.3.2 different methods to swap an application out of memory. For the set of applications that are running, their processes are co-scheduled by each node's operating system.

3.3.1 Network Management for Application Switching To stop an application, the network status has to be saved. We use the Chandy–Lamport algorithm to flush the network before the application is stopped. The checkpoint scheduler and the communication daemons are designed to save the network state. The checkpoint scheduler requests every communication daemon to take a global snapshot by sending them a tag. On the reception of a tag, a daemon stops the computing process, and then sends the tag in every communication channel (including the checkpoint scheduler). When the checkpoint scheduler has received a tag from every communication daemon, the network flush is achieved.

3.3.2 Memory Management for Application Switching

3.3.2.1 SIGSTOP/SIGCONT Policy, System Memory Management In previous implementations of gang scheduling, each process of an application is stopped by the SIGSTOP UNIX signal. In the case of large memory consumption, memory sharing is managed by the operating system's swap policy. As running applications were swapped out to 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 the 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

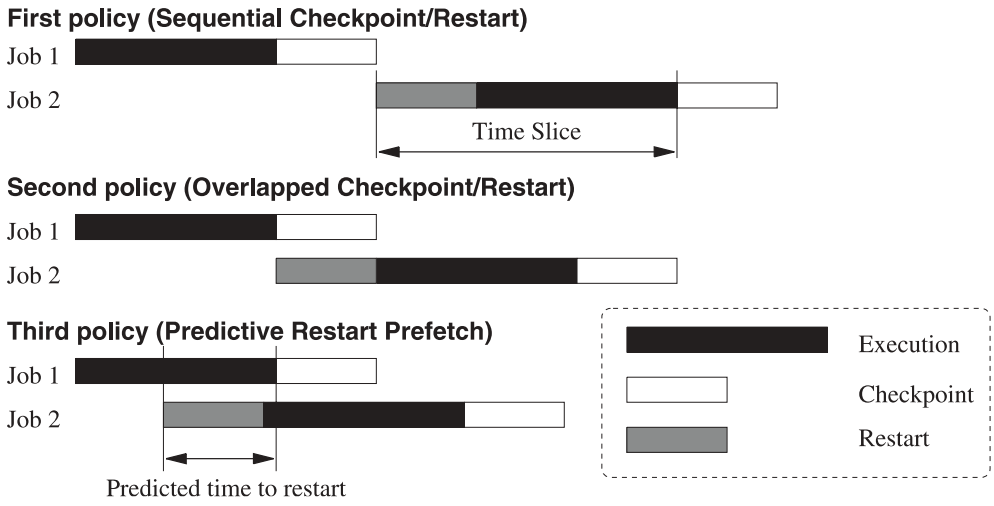

Fig. 4 Policy for checkpointing a job and restarting the next scheduled one on the same node.

faults. Thus, we prefer to use a checkpoint based technique, whose overhead is well bounded.

3.3.2.2 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 the local disk, thus freeing all memory it uses. The complete memory of the process to be scheduled in the next time slice is reloaded from the checkpoint, and thus no page fault occurs during the time slice. Incremental checkpoint related techniques may be used to improve checkpoint performances. This memory management is equivalent to the aggressive paging out technique described in Ryu, Pachapaurkar, and Fong (2004).

Many policies may be used to overlap checkpoint, restart, and computations. Figure 4 presents the three techniques implemented in the comparison framework. The first method (called sequential checkpoint/restart) does not overlap checkpointing and restarting. It avoids loading 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 application 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 5.2.

3.3.3 Paired Hybrid Scheduling In the context of pure gang scheduling, Wiseman and Feitelson (2003) prove that scheduling two applications with very different communication/computation ratios during the same time slice improves performance greatly. This optimization can be used in the context of hybrid scheduling of heterogeneous applications, by building the co-scheduled subset with consideration to the communication/computation ratio of the applications, and trying to avoid creating homogeneous subsets.

4 Common Framework

In this paper, we focus on comparing and mixing two scheduling methods for MPI application time sharing. The first is based on the ability to stop and restart a set of

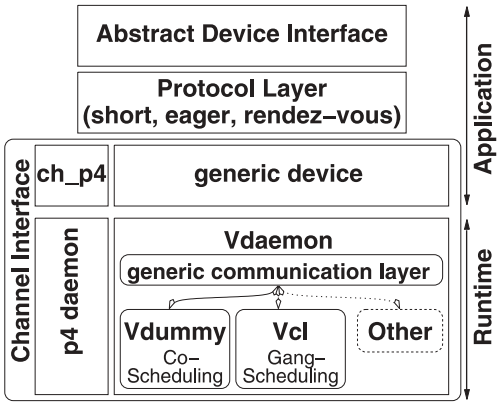

Fig. 5 Architecture of MPICH-V compared to architecture of MPICH-P4.

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 method 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. Its main focus is comparison of different types of fault-tolerant protocols for MPI. Deviating from this main purpose, we developed two non-fault-tolerant protocols. One includes a distributed checkpoint facility based on the Chandy–Lamport algorithm (Vcl; Bouteiller et al. 2003; Lemarinier et al. 2004), and another implements 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 Lemarinier et al. (2004). Figure 5 compares the architectures, while Figure 7 shows the ping-pong performance comparison between the MPICH-V framework and the reference implementation, and validates the checkpoint enabled MPICH–Vcl version performance compared to the standard MPICH–Vdummy version.

MPICH-V is based on the MPICH library (Gropp et al. 1996), 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 of 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 the 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 us 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 sit on top of these low-level drivers. So, this is one of the most relevant layers 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 these have been slightly modified to focus on the scheduling of MPI applications, while some have been added, specifically to manage sets of applications.

4.1 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 its own dispatcher.

4.2 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). The asynchronism of communication is delayed to the daemon.

4.3 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 component. Every send or receive

operation is asynchronous. Thus, a communication is not blocked by another slower one. On the contrary, the communication across the interprocess 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).

4.3.1 Generic Communication Daemon Daemons implement a generic communication layer to provide all the point-to-point communication routines between the different types 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 5).

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.

4.4 Checkpoint Scheduler

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

4.5 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 components of each application. The application deployment itself is performed by the dispatcher of this application (figure 6).

The number n of jobs to run simultaneously is given as a parameter of the master scheduler. The master scheduler associates a time slice with 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.

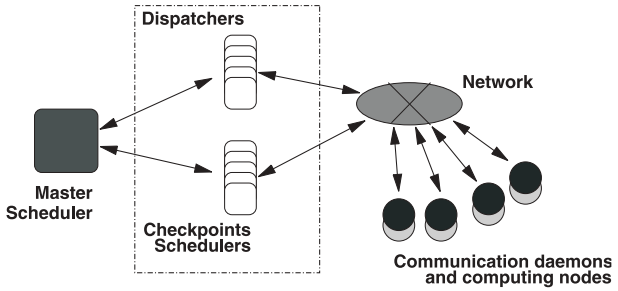

Fig. 6 Typical deployments of MPICH-V with many applications.

The master scheduler implements various policies for stopping/restarting applications. It is possible to co-schedule k applications of a set of n. To perform set context switching, three checkpoint/restart overlap policies are implemented. These policies are detailed in Section 3.3.2.

5 Performance Evaluation

5.1 Experimental Conditions

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

All these nodes use Linux 2.4.21 as the 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 five times and we present a mean of these.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. We use the NetPIPE (Snell, Mikler, and Gustafson 1996) utility to measure the 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 set of kernels and applications of the Numerical Aerodynamic Simulation (NAS) Parallel Benchmark suite (Bailey et al. 1995), written by the National Aeronautics and Space Administration (NASA) Numerical Aerodynamic Simulation NAS research center to test high performance parallel computers. These benchmarks cover a large panel of communication schemes:

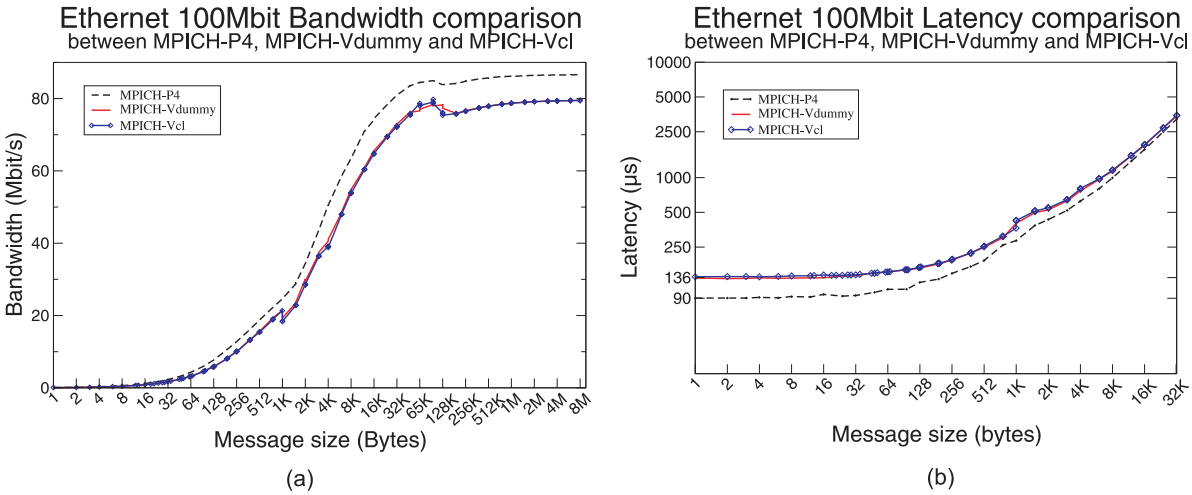

Fig. 7 Bandwidth and latency comparison between MPICH-P4, MPICH-Vdummy, and MPICH-Vcl on the Fast-Ethernet network.

each benchmark tests a particular communication scheme, and communication computation ratio. CG benchmark presents heavy point-to-point latency driven communications; BT benchmark presents large point-to-point messages, and communications overlapped by computation. Each process of a BT benchmark uses 175 MB of memory for class C on 25 nodes, 135 MB for class B on nine nodes, and each process of CG class C on eight nodes uses 157 MB. A node has 1 GB of memory (900 MB of user space memory). Thus, up to five simultaneous applications fit in physical memory whatever the application we use. Moreover, with seven or more simultaneous applications, swapping is always needed.

As we compare scheduling algorithms in terms of both performance and fairness, we propose to use the following trade-off metric to measure performance: the slowdown between concurrent application set makespan and time to perform an according number of sequential executions plus the standard deviation of individual concurrent execution time over the average concurrent execution time:

$$\frac{t_{makespan}}{n.t_{sequential}} + \frac{\sqrt{\Sigma[(t_{concurrent} - \overline{t_{concurrent}})^2/n]}}{\overline{t_{concurrent}}}.$$

5.2 Checkpoint/Restart Overlap Scheduling Policy

Figure 8 presents the performance of NAS BT class C benchmark on 25 nodes for the three application context switching policies of checkpoint/restart presented in Section 3.3.2.2. In Figure 8(a), two concurrent applications are gang scheduled. In Figure 8, five applications of 10 are run simultaneously, thus filling physical memory.

When physical memory is not exhausted by computing applications, as expected, the more overlap reached, the better the performance. Thus, the overlapped checkpoint/restart method performs better than sequential checkpoint/restart, and the predictive restart prefetch method performs better than the other two. 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 enough applications to exhaust physical memory, the overlapping strategies do not perform as well. The finishing application remains in memory during its checkpoint, thus inducing memory swapping for the 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 being filled.

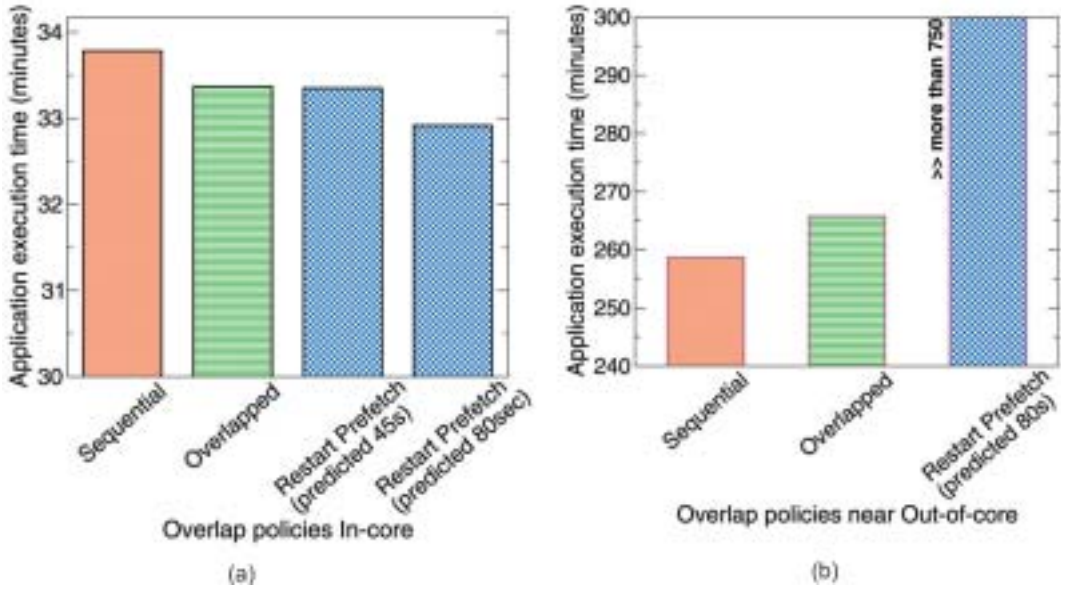

Fig. 8 Comparison of checkpoint/restart application context switching policies, using the NAS benchmark BT class C on 25 nodes on Ethernet performance criteria: (a) two hybrid scheduled BT.C.25, one running at a time (in-core); (b) 10 hybrid scheduled BT.C.25, five running at a time (near out-of-core).

5.3 Scheduling Techniques Performance Comparison

Figure 9 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 seven simultaneous applications, the overall throughput is decreasing, and for nine simultaneous applications, the makespan is 20 times the sequential batch scheduling makespan. Table 1 explains this result, presenting the average number of major page faults per minute for seven and nine simultaneous applications using co-scheduling. On the one hand, for seven 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 minute), while another suffers from more than 1245 page faults per minute. On the other hand, for nine co-scheduled applications, all applications hit the swap equally (standard deviation is less than 47.1 for an average of 507.4 page faults per minute). This illustrates 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 achieves good performance

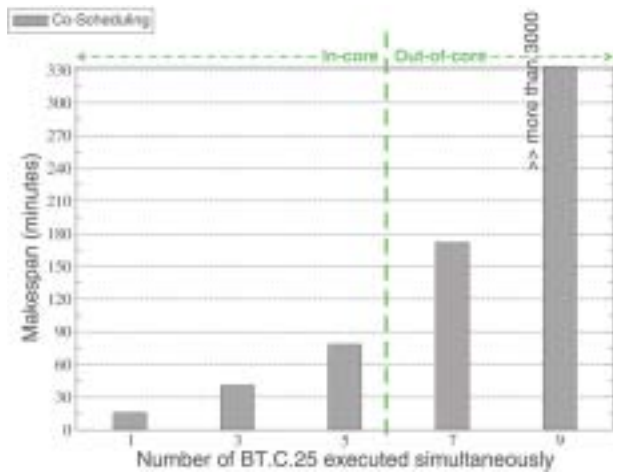

Fig. 9 Makespan for *n* simultaneous BT class C 25 nodes using co-scheduling.

by scheduling more time for the in-core applications, at the cost of serializing executions. When memory occupa-

Table 1
Page fault statistics for simultaneous BT class C on 25 nodes.

Number of applications	Average number of major page faults for all nodes of each application, per minutes									Average page faults per min 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**

tion 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 seven simultaneous applications. For nine 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 10 presents the computation time of *n* simultaneous NAS benchmarks, using co-scheduling, gang scheduling or hybrid scheduling. Hybrid scheduling relies on the operating system scheduler for executing a set of five 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 a set of applications context switch. Time slices are 900 s.

For the CG benchmark (Figure 10(a)), the co-scheduling method reaches good performance up to using about three times the physical memory (2800 MB). Comparing to the BT benchmark results (Figure 9), this shows that the point of inefficiency of co-scheduling is tightly related to the pattern of memory accesses. The application often uses the same pages, reducing the number of page faults. However, for these two applications, the huge impact on co-scheduling of the kernel's aggressive swapping policy is observed. When co-scheduling is outperformed, gang and hybrid scheduling techniques reach the same performance. This shows that there is almost no benefit of co-scheduling a subset of applications for this benchmark. In

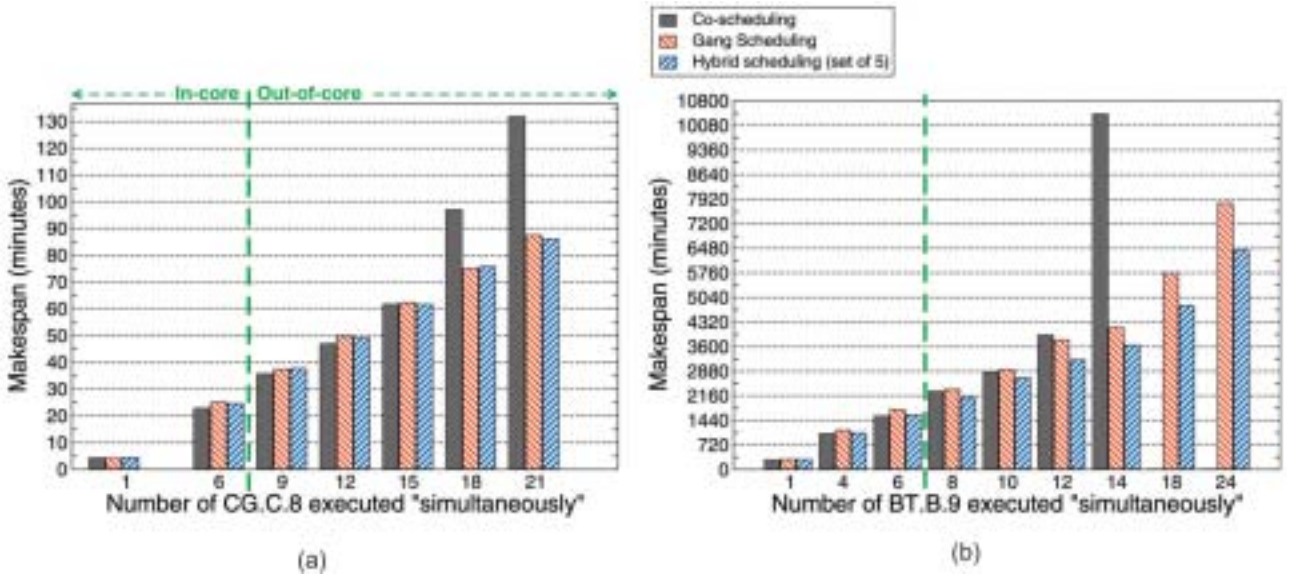

Fig. 10 Makespan for *n* simultaneous applications using co-scheduling, gang scheduling or hybrid scheduling: (a) CG class C 8 nodes; (b) BT class B 9 nodes.

Table 2
Average slowdown using co-scheduling, gang scheduling or hybrid scheduling, compared to sequential batch scheduling.

		Co-sched	Gang sched	Hybrid sched
CG	in-core	0.92	1.02	0.99
	out-of-core	1.3	1.02	0.99
BT	in-core	0.9	1.02	0.90
	out-of-core	1.9	1.07	0.90

CG, communication prevails, and thus the bandwidth is divided between co-scheduled applications.

Figure 10(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 however many fit 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 type of application, hybrid scheduling performs 17% better than gang scheduling.

Table 2 compares the relative slowdown of co-scheduling, gang scheduling and hybrid scheduling compared to ideal (overhead free) sequential batch scheduling. For the CG benchmark, gang and hybrid scheduling perform equally to sequential scheduling. Co-scheduling shows a 6–8% performance improvement until we reach out-of-core computing, where it shows a 30% performance penalty. For the BT benchmark, in-core, co-scheduling and hybrid scheduling give a 10% improvement compared to sequential scheduling, while gang scheduling suffers from a 2% overhead. When out-of-core memory is used, co-scheduling performance is very poor, and gang-scheduling suffers from a 7% slowdown. In the same context, hybrid scheduling produces a 10% improvement compared to sequential scheduling. For every benchmark, the overall throughput of hybrid scheduling is equal or up to 10% better than sequential scheduling.

5.4 Case Study for Heterogeneous Applications

Focusing on the case of heterogeneous sets of applications, let us consider a total of n applications, where $n/2$ are communication bound (CG.C.8 in this experiment) and $n/2$ are computation bound (BT.B.9). With regard to available physical memory, up to five of these applications may be co-scheduled during each time slice. However, in order to have the same number of computing and communicating applications, we restricted the subset size to four in this experiment. On the one hand, for homogeneous hybrid scheduling, the construction of each subset ensures that only four identical applications are running during each time slice. Thus, half of the time slices are dedicated to only communicating applications, and half to computing applications. On the other hand, for paired hybrid scheduling, all subsets are identical and composed of two communicating and two computing applications. Figure 11 presents homogeneous hybrid scheduling compared to paired hybrid scheduling. Whatever the total number of applications, considering the different communication/computation ratio of applications improves the hybrid scheduling performance by 2.7%. Compared to the 17% performance improvement between gang and hybrid scheduling, this illustrates that most of the communication/computation overlap is due to co-scheduling, and only a small part is related to heterogeneousness.

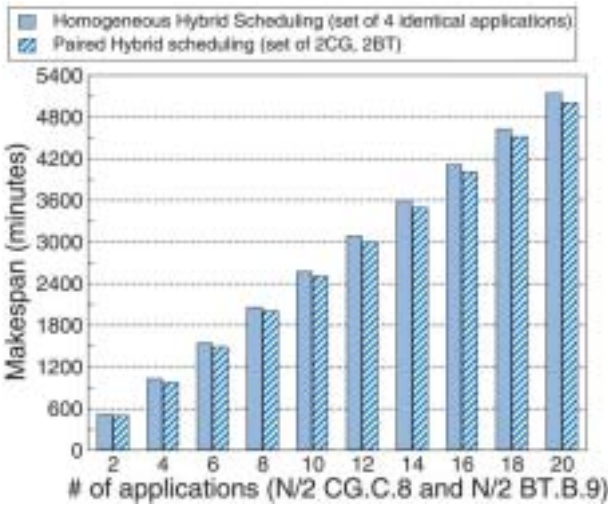

Fig. 11 Makespan for *n* simultaneous heterogeneous applications using homogeneous hybrid scheduling or paired hybrid scheduling.

6 Conclusion

In this paper, we compare several scheduling techniques for the grid. Some are well known, such 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 approaches, which we call hybrid scheduling.

Hybrid scheduling splits 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 the 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 switching.

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. We show that hybrid scheduling can benefit from scheduling simultaneously applications with different computation/communication ratios. Compared to sequential batch scheduling, hybrid scheduling reaches 10% better or equal throughput depending on the type of application, 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 the 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.

Acknowledgments

The MPICH-V project is founded by the French ministry of research through the "ACI Grid" initiative. We would like to thank Larry Carter, from the University of San Diego, who has kindly helped us to improve the overall quality of the paper with his thoughtful reading.

Author Biographies

Aurelien Bouteiller is a Ph.D. student in the Cluster and Grid group of the Laboratoire de Recherche en Informatique (LRI), Universite Paris-Sud, and is a member of the Grand-Large team of INRIA. He obtained a Master degree in parallel computer science in 2002 from Universite Paris-Sud. He contributes to the MPICH-V project, a fault-tolerant MPI implementation comparing different fault-tolerant protocols. His research interests include high performance fault-tolerant MPI and checkpoint optimizations and performance evaluation.

Hinde Lilia Bouziane obtained a Master degree in computer science in September 2004 from Universite Paris-Sud. She spent five months in the Grand-Large research team of INRIA working on hybrid scheduling of MPI applications for clusters. In November 2004, she started a Ph.D. in the PARIS research team of INRIA, located at IRISA (Rennes, France). Her research interests concern advanced programming models for Grids, involving software component and workflow models.

Thomas Herault is an associate professor at the Universite Paris-Sud. He defended his Ph.D. on the mending of transient failure in self-stabilizing systems under the supervision of Joffroy Beauquier. He is a member of the Grand-Large INRIA team and works on fault-tolerant protocols in large-scale distributed systems. He contributes to the MPICH-V project, whose goal is to evaluate and compare fault-tolerant protocols for MPI.

Pierre Lemarinier obtained a Master degree in computer science in 2002 from the Universite Paris-Sud. He is a Ph.D. student in the parallelism team and the Grand-Large team of the LRI laboratory of Universite Paris-Sud. His research interests include fault-tolerant protocol conception and validation and MPI implementations (MPICH-V).

Franck Cappello holds a Research Director position at INRIA, after having spent eight years as a CNRS researcher. He leads the Grand-Large project at INRIA and the Cluster and Grid group at LRI. He has authored more than 50 papers in the domains of high performance programming, desktop grids, and fault-tolerant MPI. He is an editorial board member of the *International Journal on GRID Computing* and a steering committee member of the IEEE/ACM CCGRID. He organizes annually the Global and Peer-to-Peer Computing workshop. He has initiated and heads the XtremWeb (desktop grid) and MPICH-V (fault-tolerant MPI) projects. He is currently involved in two new projects: Grid eXplorer (a grid emulator) and Grid5000 (a nationwide experimental grid testbed).

References

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

Bouteiller, A., Lemarinier, P., Krawezik, G., and Cappello, F. 2003. Coordinated checkpoint versus message log for fault-tolerant MPI. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, December, IEEE Computer Society Press, Los Alamitos, CA.

da Silva, F. A. B. and Scherson, I. D. 2000. Improving parallel job scheduling using runtime measurements. *Proceedings of*

the 6th Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'00), Cancun, Mexico, May, pp. 18–38.

Feitelson, D. G. and Jette, M. A. 1997. Improved utilization and responsiveness with gang scheduling. *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'97)*, Geneva, Switzerland, April, D. G. Feitelson and L. Rudolph, editors, Vol. 1291, Springer-Verlag, Berlin, pp. 238–261.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. High performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6):789–828.

Henderson, R. L. 1995. Job scheduling under the portable batch system. *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'95)*, Santa Barbara, CA, April, pp. 279–294.

Hori, A., Tezuka, H., Ishikawa, Y., Soda, N., Konaka, H., and Maeda, M. 1996. Implementation of gang-scheduling on workstation cluster. *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'96)*, Honolulu, HI, April, D. G. Feitelson and L. Rudolph, editors, Springer-Verlag, Berlin, pp. 126–139.

Hori, A., Tezuka, H., and Ishikawa, Y. 1998. Overhead analysis of preemptive gang scheduling. *Lecture Notes in Computer Science*, Vol. 1459, Springer-Verlag, Berlin, pp. 217–230.

Lee, W., Frank, M., Lee, V., Mackenzie, K., and Rudolph, L. 1997. Implications of I/O for gang scheduled workloads. *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'97)*, Geneva, Switzerland, April, D. G. Feitelson and L. Rudolph, editors, Vol. 1291, Springer-Verlag, Berlin, pp. 215–237.

Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., and Cappello, F. 2004. Improved message logging versus improved coordinated checkpointing for fault-tolerant MPI. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2004)*, San Diego, CA, September, IEEE Computer Society Press, Los Alamitos, CA.

Parsons, E. W. and Sevcik, K. C. 1997. Implementing multiprocessor scheduling disciplines. *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'97)*, Geneva, Switzerland, April, D. G. Feitelson and L. Rudolph, editors, Vol. 1291, Springer-Verlag, Berlin, pp. 166–192.

Ryu, K. D., Pachapurkar, N., and Fong, L. L. 2004. Adaptive memory paging for efficient gang scheduling of parallel applications. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, April.

Snell, Q., Mikler, A., and Gustafson, J. 1996. Netpipe: a network protocol independent performance evaluator. *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, Washington, DC, June.

Strazdins, P. and Uhlmann, J. 2004. A comparison of local and gang scheduling on a Beowulf cluster. *Proceedings of the 2004 IEEE International Conference of Cluster Computing*, San Diego, CA, September, IEEE Computer Society Press, Los Alamitos, CA, pp. 55–62.

Tezuka, H., Hori, A., Ishikawa, Y., and Sato, M. 1997. PM: an operating system coordinated high performance communication library. *Proceedings of the International Conference and Exhibition on High Performance Computing and Networking*, Vienna, Austria, April, Springer-Verlag, Berlin, pp. 708–717.

Wiseman, Y. and Feitelson, D. G. 2003. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems* 14:581–592.