Paired Gang Scheduling*

Yair Wiseman§
School of CS & Engineering
Hebrew University
and Dept. of Computer Science
Bar-Ilan University
wiseman@cs.biu.ac.il

Dror G. Feitelson
School of CS & Engineering
Hebrew University
Jerusalem
Israel
feit@cs.huji.ac.il

Abstract

Conventional gang scheduling has the disadvantage that when processes perform I/O or blocking communication, their processors remain idle, because alternative processes cannot be run independently of their own gangs. To alleviate this problem we suggest a slight relaxation of this rule: match gangs that make heavy use of the CPU with gangs that make light use of the CPU (presumably due to I/O or communication activity), and schedule such pairs together, allowing the local scheduler on each node to select either of the two processes at any instant. As I/O-intensive gangs make light use of the CPU, this only causes a minor degradation in the service to compute-bound jobs. This degradation is more than offset by the overall improvement in system performance due to the better utilization of the resources.

Keywords: Gang Scheduling, Job Mix, Flexible Resource Management

1 Introduction

Scheduling parallel jobs on a parallel supercomputer consists of deciding which processors to use for each parallel job, and when to run it. A simple first-come-first-serve (FCFS) approach keeps waiting jobs in a queue in the order that they arrived. The scheduler then tries to find free nodes for the first job in the queue. If there aren't enough free nodes, the job will have to wait until nodes become available when some job that is currently running terminates. When this happens, the needed nodes are allocated and the queued job starts to run. Using FCFS prevents starvation of jobs which use a large number of nodes, but may cause nodes to be left idle when waiting for more nodes to become available. This can be reduced by using backfilling, in which jobs may be selected out of order provided this will not delay the execution of jobs which have a higher location in the queue [13].

Another approach, called gang scheduling [14], slices the time in a round-robin manner. The jobs are packed into a matrix. Each column in the matrix represents a node and each row represents a time slot. Jobs assigned to different rows are executed one after the other. One of the problems

*A preliminary version appeared in JPDPS-2001.
§Research supported by a Lady Davis Fellowship.

with this approach is fragmentation: processors may be left idle if nothing is assigned to them in a certain slot. Several ideas for relaxations of gang scheduling that reduce this waste have therefore been proposed [11, 7, 22, 1]. Our work is also a relaxation of strict gang scheduling, which is called *Paired Gang Scheduling* [23]. All of these improvements remove the requirement that all processes of a given job always run simultaneously on different nodes. There are also other types of improvements, e.g. the use of backfilling to handle the queue of jobs being packed into the gang scheduling matrix [24, 17].

Gang scheduling enables processes in the same job to run at the same time. This leads to better performance for compute-bound communicating processes [6]. However, I/O-bound processes cause the CPUs to be idle too much of the time, while there are other processes which can run. At the same time, the effect on the disk performance is the opposite: I/O-bound processes keep the disks busy, while compute-bound processes leave them idle. Indeed, it is non-trivial to balance the use of these resources in applications that have large computation and I/O requirements [15].

The core idea of gang-scheduling is assigning as many processors to a job as are required at the same time. Such an assignment allows the job to avoid blocking processes while they wait for the completion of communications with other processes, because of two reasons:

1. It is guaranteed that the awaited process is running and making progress, so it makes sense to wait for it.
2. There is nothing else to run on the processor (all the job's processes are assigned).

By contradistinction, when one assigns more than one process to each processor, it can lead to situations where one process needs to wait for another process to be executed again, because it is not currently running. Thus gang scheduling strives to maximize the performance of the current job, at the possible expense of overall system utilization.

The alternative to gang scheduling is to use local scheduling independently on each node of the machine. The local scheduler can use round-robin, or a priority-based algorithm such as the one used in Unix. In such a system, a process that needs to wait for another should block, because the two conditions listed above are negated: the awaited process is probably not running, and there are other better things to do with the processor rather than busy wait. In other words, local scheduling has the pretension of emphasizing overall system utilization, at the possible expense of jobs that need to perform a lot of communication. However, the extra context switches induced by fine grain communication can lead to inefficiency and reduced effective utilization [6].

If the characteristics of each gang are known, this can be exploited in order to keep both the CPU and the I/O devices busy. This paper presents the idea of matching *pairs* of gangs, one compute-bound and the other I/O-bound. The rationale for such matching is that these gangs will hardly interfere in each other's work, as they use different devices. Therefore they will "feel" that they work alone in the system. If the I/O operations' time is not negligible relative to the CPU time, such an overlap of the I/O activity with CPU work can be efficient [16]. The concept is illustrated in Figure 1. It is similar to the idea of symbiotic job scheduling proposed for SMT processors [21].

Paired gang scheduling tries to be a compromise between the above two schemes, with the goal of utilizing the system resources well without causing interference between the processes of competing jobs. It enjoys the best from the two worlds. On one hand the processes won't have to wait much because a process which occupies the CPU most of the time will be matched with a process that occupies an I/O device most of the time, so they will not interfere with each other's

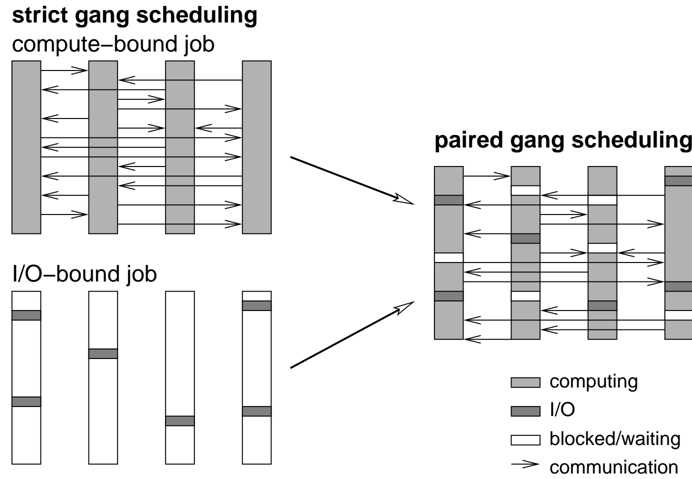

Figure 1: *Paired gang scheduling: by running pairs of complementary jobs, resource utilization is improved without undue interference. Vertical bars represent processes, with time flowing downwards.*

work. On the other hand, the CPU and the I/O devices will not be idle while there are jobs which can be executed.

The rest of the paper is organized as follows. Section 2 presents the model of paired gang scheduling and the algorithms used in its implementation. Section 3 describes the platform of the realization and what changes were required to support paired gang scheduling. Section 4 evaluates the concept of paired gang scheduling. It reports on the experiments that have been conducted and the workloads that were used. Finally, section 5 summarizes the results.

2 Paired Gang Scheduling

The problem of how I/O-bound jobs affect system performance under gang scheduling is discussed by Lee et al. [11]. They suggest a method of varying the time quantum which is given to processes according to their characteristics. The proposal of this paper is leaving the time quantum as a constant, but dispatching a pair of processes in each time quantum. The matching of processes is based on a prediction of what their CPU utilization will be in the next quantum.

2.1 Framework for Scheduling

The framework for paired gang scheduling is depicted in Figure 2. We use a centralized gang scheduler, as is used in many conventional implementations [4, 9, 5, 8]. This is a user-level daemon process that runs on a distinguished node in the cluster. It maintains information in an Ousterhout matrix [14], in which rows denote scheduling slots and columns represent processors. In strict gang scheduling, one row is selected each time. Each node scheduler is then directed to schedule the process in the respective cell of the chosen row. This is typically done by making only this process runnable, while the others are blocked (in Unix implementations this is typically done by sending them a SIGSTOP signal).

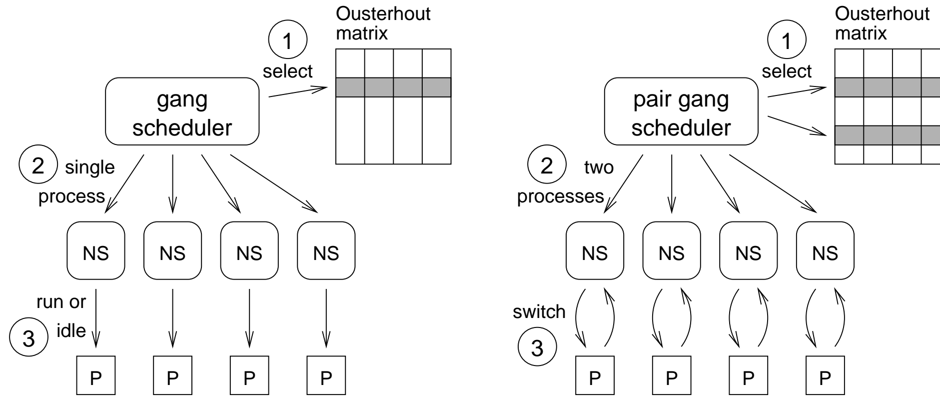

Figure 2: *Framework for paired gang scheduling (right), as opposed to strict gang scheduling (left). NS = node scheduler; P = processor.*

For paired gang scheduling, two rows are selected. In this case two processes are left runnable on each node, and their scheduling is done at the discretion of the local scheduler. In essence, the paired gang scheduler uses its power over the local workload to generate a good job mix with two complementary processes. Similar mixes are generated on all nodes by virtue of the homogeneity of processes in the same row of the matrix.

2.2 Measuring CPU Utilization

The idea is to match jobs that use the CPU with those that use I/O devices (including the network). This seems to imply that we need to measure both CPU utilization and I/O activity for each job. However, as long as we bundle all the I/O activity into a single class, I/O and CPU usage are actually complementary. A process that performs I/O is blocked, and is prevented from utilizing the CPU until the I/O completes. Thus it is enough to measure the CPU utilization, and postulate that the I/O utilization is the remaining time. Indeed, this way of measuring even works for applications that use non-blocking I/O, thus boosting their CPU utilization.

Note that by using this definition paging is included in I/O, because the operating system blocks processes that are waiting for a page fault to be serviced. While this is not strictly a characterization of the inherent I/O activity of the process, it does characterize the I/O activity under the current system conditions (physical memory installed and competing jobs that also use memory). Thus it is appropriate to include it.

The measured CPU utilization can be used directly. But consider a case where two gangs are matched because they have a low CPU utilization, and then this changes and they both become compute-bound. Due to the fact that they are running together they will by necessity achieve CPU utilizations that are together bounded by 100%, so the scheduler will think that it is appropriate to continue and schedule them together. A possible workaround is to replace the actual CPU utilization by the effective utilization, defined as the utilization *out of what is left over by the other process*. Specifically, if processes x and y have measured utilizations $u(x)$ and $u(y)$, respectively, their effective utilizations will be

$$u_{eff}(x) = \frac{u(x)}{1 - u(y)} \quad \text{and} \quad u_{eff}(y) = \frac{u(y)}{1 - u(x)}$$

However, this would lead to misleading results if the true CPU utilizations are together slightly less than 100%. A better solution is therefore to monitor *changes* in the CPU utilization. When the utilization of a process that shares the processor grows, it can then be speculatively assigned a utilization of 100%, thus forcing it into a slot by itself. This then provides a better test of the true CPU utilization. If it fails the test (that is, if its utilization is actually much lower than 100%), it will be matched again with another process the next time it is scheduled.

2.3 Predicting CPU Utilization

The characteristics of a process can be assessed by looking at its history. A reasonable guess is that if a process had a high/low CPU utilization in its past, the process will continue to have a high/low CPU utilization in its future, respectively. In [19] Silva et al. introduce a way for classifying processes using fuzzy sets. An extension of their idea can be obtained by using Markov chains. A Markov chain can be useful in investigating the process' history. A simple Markov chain can be the average of the last N CPU utilization values as measured in the last N time slices. A more complex Markov chain can give more weight to the recent slices, while giving a reduced weight to the far history. In [18] the authors suggest the formula:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Where t_n is the actual length of nth CPU burst, τ_n is the predicted value for that CPU burst, and α is a coefficient having a value between 0 and 1.

Their method was used for predicting the length of the next CPU burst, but it can be trivially adapted for predicting utilization. However, this method never forgets any CPU burst which was in the process' history. Although the influence of a very old burst becomes very small, it is still there. A Markov chain of the last N CPU bursts can totally forget the old bursts, which can be meaningless. Moreover, this formula considered just one process. We would like to have a formula which will take into account all of processes that make up a parallel job.

The Markov Chain which was used in our test is:

$$\tau_n = \sum_{i=1}^{4} \left[\frac{(5-i)}{10} \cdot \sum_{j=1}^{m} \frac{t_{n-i,j}}{m} \right] \quad (1)$$

Where:

- n is the serial number of the next time quantum for this job.
- τ_n is the predicted CPU utilization in the next time slice.
- $t_{i,j}$ is the actual CPU utilization in the i time slice on the j processor.
- m is the number of processors used by the job.

The first sum is over the 4 last CPU quanta, and gives them linearly decreasing weights. The inner sum produces the average for the job's processes on the different processors.

At the beginning, before any concrete data is available, new jobs will be postulated to have 100% CPU utilization. This initialization will force each job to be executed separately in its first time quantum, and gives the system a chance to measure the true utilization. If another initialization

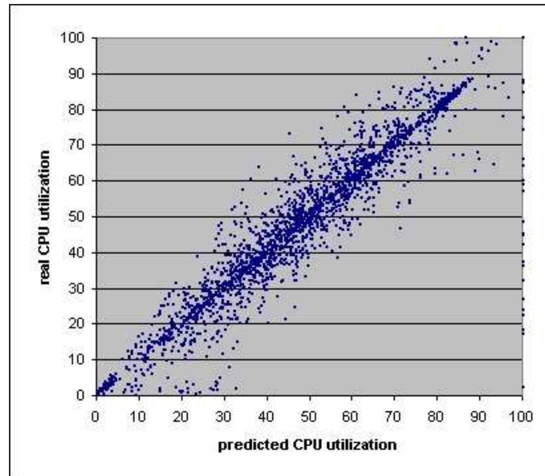

Figure 3: *Predicted CPU utilization compared with real CPU utilization.*

were to be used, there is a danger that two computed-bound jobs execute together, leading to an erroneous measurement of less than 50% CPU utilization. Thus we start with a speculative value of 100%, as we also suggest to do after a sharp increase is detected. Figure 3 shows the correlation between the predicted CPU utilization and the real CPU utilization when running heterogeneous applications as described in Section 4.2.2. The dots at the right are new jobs that were speculatively predicted to have 100% utilization.

2.4 Matching Pairs of Jobs

The data is represented in percents: 100% means all the time quantum was dedicated to CPU activity in this process, while 0% means none of the time quantum was dedicated to CPU activity. The master collects the data from all active slaves. Then, the master calculates the average CPU utilization of all processes in the job. This is calculated anew at the end of each quantum in which the job ran.

Scheduling is based on an Ousterhout matrix [14], with the jobs in successive slots scheduled in turn in a round-robin manner. But as the job in the next slot is considered, the master will look for a match for this job, and schedule *both* of them. Let us assume the job to be scheduled now (the one in the next slot) has a predicted x% CPU utilization as calculated by formula (1). The match for this job will be a job whose predicted CPU utilization y, again according to formula (1), satisfies

$$x + y + m < 100\%$$

where m is a safety margin (we used $m = 1\%$). Such a match can keep the CPU busy on one hand, but will not cause the processes to wait for the CPU to be freed, on the other hand.

A naive approach could do the matching dynamically at each context switch, based on the most recent data available, using a best-fit searching procedure. Thus, if several jobs tie as the best match, the last one will always be chosen. Moreover, such a matching method is unfair. Suppose we have a lot of I/O-bound jobs with very low CPU utilizations, e.g. 5% or less. If we have two other jobs having 80% and 81% CPU utilizations, the 81% job will be the ideal match for each of the low CPU jobs, so the 81% job will always be selected, giving it much more CPU time than its

```
sort the jobs according to their CPU time
p_start points to the beginning of the sorted vector
p_end points to the end of the sorted vector
while p_start < p_end
   while (CPUutil(p_start) + CPUutil(p_end) + m ≥ 100%) and (p_start < p_end)
      match p_end with NULL
      p_back=p_start - 1
      while p_back still points to a valid job
         if CPUutil(p_back) + CPUutil(p_end) + m < 100%
            match p_back and p_end
            break
         p_back = p_back - 1
      p_end = p_end - 1
   if p_start < p_end
      match p_start with p_end
      p_start = p_start + 1
      p_end = p_end - 1
```

Figure 4: *The matching algorithm (indentation is used to delimit blocks).*

peer which has just 80% CPU utilization. In order to be more fair, we have used another matching method, in which matching is performed once for each round of the whole matrix. The matching algorithm is described in Figure 4. The idea is to sort the jobs according to their CPU utilization, and match jobs from both ends. Thus each job will be matched with a distinct other one if a possible match exists. Only jobs which didn't match any other unmatched job, will be matched to a matched job. As a result, the waiting queue will be smaller and the jobs will get more uniform service. If a job needs exclusive control of the system's resources (e.g. for running benchmark), a flag can be set in order to indicate to the system that this job must not be matched with another job.

2.5 Extension to General Gang Scheduling

The above description was based on matching single jobs. But in the general case more than one job may be allocated to the same slot. If this is the case, the above will be generalized to matching slots instead of jobs. This is achieved by modifying the formula to

$$\max(x) + \max(y) + m < 100\%$$

where $\max(x)$ and $\max(y)$ are the maximal CPU utilizations for jobs in slots x and y, respectively.

It is desirable that all jobs in a slot will be homogeneous. Sometimes, an exceptional job can cause a problem. A given slot can be a perfect match to another, but the scheduler will not match them because an exceptional job has much higher CPU utilization than the other jobs in the same slot. In order to prevent such cases, we would like to have homogeneous slots, in which all jobs have similar characteristic. We therefore check the characteristics of each running job on every context switch. A job J in slot s which doesn't satisfy $CPUutil(J) < \min(s) + M$ (where M

is the allowed range within the slot), will be removed from the slot. A search will be invoked in order to find another slot for the removed job. If no such slot is found, the job will be put in a new slot. In our tests we have used $M = 20\%$. In effect, the range of each slot is then bounded as in $\max(slot) - \min(slot) \leq 20\%$.

2.6 Alternate Scheduling

Alternate Scheduling [3] can improve gang scheduling. Using this method we look for idle nodes in a slot. If we find some, we will try to execute a job or even several jobs which need these nodes or part of them. When using paired gang scheduling, we should take into account the characteristic of the alternate-scheduled jobs. In other words, when a job is assigned for alternate scheduling in a slot, it must match the characteristics of other jobs that are already assigned to this slot. If its utilization does not fall within the band of $M = 20\%$ allowed, alternate scheduling will not be used. However, this does not mean that the nodes will remain idle: In paired gang scheduling each node is supposed to run *two* processes. If no alternates are found, only one process will run. A node will remain idle only if it is not allocated in both selected slots.

3 Implementation

3.1 The ParPar Environment

The implementation of paired gang scheduling was done in the context of the ParPar cluster [4]. The cluster has a host node which is the “master” and 16 other nodes which are the “slaves”. The master runs a daemon which controls all system-wide activity. Among other things, it decides for the slaves when they should do a context switch. The master also maintains the gang scheduling matrix. When the master decides on a context switch, it sends a message to the slaves which job should be executed instead of the current job. The slaves stop the current job and start executing the new job as ordered by the master.

Each node is an independent PC with its own processor, memory, and disk. The slaves and the master are connected by a switched Ethernet. The master sends broadcast messages using UDP, and slaves reply using TCP [10]. The communication of applications is performed using TCP/IP. The cluster is also equipped with a Myrinet, but this was not used as explained blow.

3.2 Modifications to Support Paired Gang Scheduling

While the original ParPar software includes support for gang scheduling, in paired gang scheduling things are a little bit more complex. The master still has to decide on context switches, but it has to take into account the characteristics of the different jobs and do the matching. The data on the characteristics of the processes comes from the slaves. At each context switch the salves send the master a message about the processes they were executing. The message contains information regarding how much of the last quantum was actually consumed by the processes as CPU time.

To get this data, the node daemon reads kernel data structures that describe different processes and their resource consumption. By comparing the readings of CPU usage at the beginning and end of the quantum, the CPU usage is obtained. While the resolution provided by the kernel is not

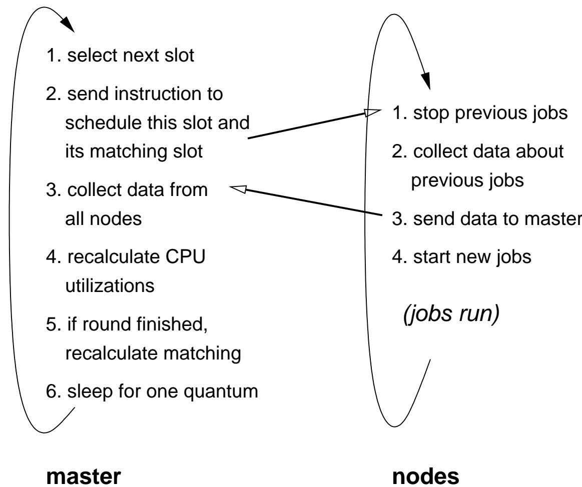

Figure 5: *Actions by the master and node daemons during a paired gang scheduling quantum.*

very high, it is adequate, especially considering that a gang scheduling quantum on ParPar is a full second.

The master daemon collects the data from all the nodes and calculates the average CPU utilization for the job. This data is then used for the matching, as described above. The way all this fits together is described in Figure 5.

The main problem with the experimental implementation is that it breaks the communication mechanisms. ParPar integrates a modified version of the FM user-level communications library for Myrinet, in which only one communication context is used, and this is replaced as part of the context switching [2]. Obviously this cannot work if two processes run at once. This can be fixed (at the cost of considerable recoding) by having two active contexts each time. However, for our evaluations, it was simpler to use conventional Unix IPC for communications, as is done in many clusters based on PVM or MPICH.

4 Evaluation

To evaluate the performance impact of paired gang scheduling, we ran several tests using the ParPar implementation described above.

4.1 Proof of Concept

4.1.1 Workload

All the experiments were based on the synthetic program shown in Figure 6, used with different parameters. For example, setting $G_{I/O} = 0$ creates a compute-bound job, whereas setting $G_{comp} = 0$ creates an I/O-bound job. Making G_{comp} and/or $G_{I/O}$ large increases the granularity (the amount of computation or I/O between barriers).

This structure of alternating compute and I/O phases is widely accepted as a reasonable model for parallel applications [15]. The barrier is implemented by all processes sending a message to process 0, and waiting for a reply. Process 0 sends the reply only after it receives a message

```
loop N times
   // compute part
   loop G_comp times
      null statement
   // I/O part
   loop G_I/O times
      open new file
      write B bytes
      close the file
   // synchronize
   barrier
```

Figure 6: *The test program used in experiments.*

from every other process in the job. In the following results, we use a mix of compute-bound and I/O bound jobs, with $G_{comp} = 2,500,000$ and $G_{I/O} = 20$ respectively; B was set to 8193. These values lead to approximately similar times for a compute barrier and an I/O barrier. Note, however, that the communication involved in barrier synchronization creates some I/O-like activity even for compute-bound processes. N was set to 250.

The proof-of-concept experiments were conducted with a mutually executed job mix. In other words, a set of jobs were all started at the same time, and each job's size was the same as the cluster size (8 nodes were used unless otherwise noted). Performance data was collected during the interval in which all the jobs ran concurrently on the system, using the original strict gang scheduling scheme or the new paired gang scheme. When the first job in the mix terminated, all other jobs were killed, and the measurements stopped. This is to ensure that the measurements indeed reflect a consistent job mix. If jobs are allowed to run to completion, the mix changes with time as some jobs terminate, and then it is harder to assign the results to the original mix.

4.1.2 Experimental Results

The metrics recorded during the runs are the rate of progress, measured in barriers completed per second, and the success rate of matching jobs to each other at runtime.

Performance Impact

To gauge the performance impact of paired gang scheduling, the performance of 5 job mixes were measured. Mix i, $i = 0 \ldots 4$, was composed of i I/O-bound jobs and $4 - i$ compute-bound jobs.

The upper part of Figure 7 shows how many barriers per second were completed by the compute-bound jobs and the I/O-bound jobs, averaging over all such jobs in each mix. The figure shows an improvement when using the paired gang scheduling as compared to the traditional gang scheduling. There is no meaningful difference when there are just compute-bound jobs in the system, because compute-bound jobs can't be matched, so the paired gang scheduling acts like the

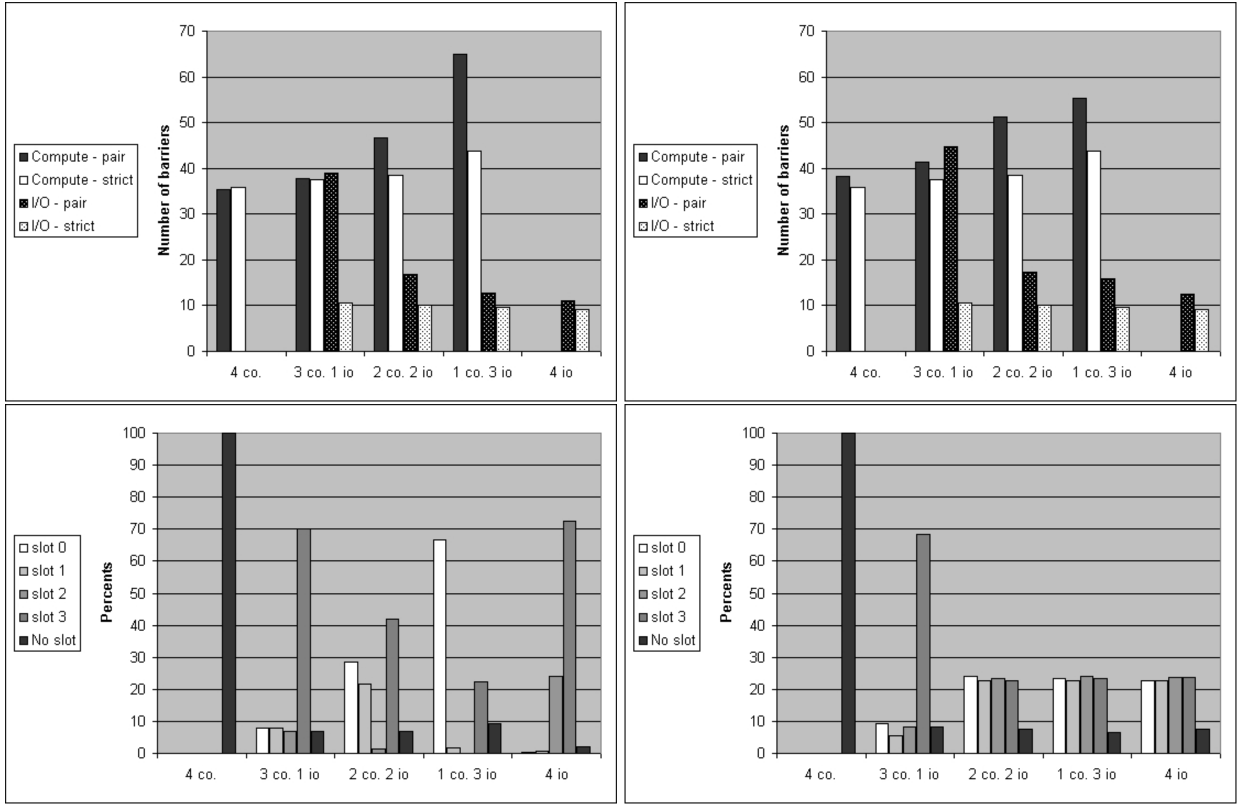

Figure 7: *Top: Performance of different jobs in each mix, as measured by barriers per second, comparing paired gang scheduling with strict gang scheduling.*
Bottom: Fraction of quanta in which jobs occupying different slots in the matrix were used for matching, in percent.
Left: using best-fit matching. Right: using the predesigned approach.

strict gang scheduling in this case. However, when there are I/O-bound jobs which can be matched with the compute-bound jobs, the progress of the jobs is accelerated in paired gang scheduling. Note that this is true both for the compute-bound jobs (especially when there are 2 or 3 I/O-bound jobs), and for the I/O-bound jobs (especially when there are 2 or 3 compute-bound jobs). Thus we find that both types of jobs benefit from the additional access to the CPU provided by being paired with another job, and this benefit outweighs the degradation suffered when another job is paired with them. The conclusion is that the interference between the matched jobs is indeed low (matching the results of [11]).

The case of a single I/O-bound job is especially interesting. When multiple I/O-bound jobs are present, they can be matched with each other, but this doesn't give a big improvement, because these jobs interfere with each other when writing to the disk. This interference causes the disk to become a bottleneck, so paired gang scheduling doesn't give a significant improvement (higher CPU allocations leading to reduced performance due to disk contention has also been observed by Rosti et al. [15]). However, when there is just one I/O-bound job and 3 compute-bound jobs, the I/O-bound job is matched with *all* the compute-bound jobs in turn. This job will therefore advance almost four times faster than when it only runs on its own, as indeed shown in the upper part of

Figure 7. This implies that the I/O-bound job achieved nearly the maximal performance possible, as if it had the system dedicated to itself — and without compromising the performance of the compute-bound jobs.

One approach to decide which jobs should run together in the same quantum, is the best-fit algorithm. The left part of Figure 7 shows the results of this algorithmic decision. Best fit always scans all 4 slots in the same order. The lower slots belong to compute-bound jobs, while the higher slots belong I/O-bound jobs. The number of jobs of each type changes in the various tests.

When there are only compute-bound jobs in the system, they all have a high CPU utilization. Therefore no matching will be done in this case, as indicated in the leftmost part of the figure.

At the other extreme, when only I/O-bound jobs are present, their CPU utilization is typically 0% (because they immediately perform an I/O operation whenever they run, and do not log any noticeable CPU time). Therefore they are all equivalent as far as the matching algorithm is concerned. As a result, the matching algorithm always chooses the last available job for matching. Thus when the jobs in slots 0, 1, and 2 are scheduled to run, the job in slot 3 is chosen as the best match. When it is slot 3's turn, the job in slot 2 is chosen. This leads to the skewed histogram in which slot 2 is chosen 25% of the time and slot 3 is chosen 75% of the time.

The intermediate cases can also be analyzed in a similar way. When there are one I/O job and three compute jobs, the I/O job is matched with all of them (75%), but they take turns being matched with it (8% each) because their CPU utilizations vary slightly. Conversely, when there is one compute job and three I/O jobs, the compute job is matched with all I/O jobs (75%), but only the last I/O job is matched with the compute job (because they all have the same 0% utilization). Finally, in the case of two jobs of each type, the compute jobs take turns being matched (around 25% each), whereas the second I/O job dominates over the first (near 50%).

An alternative is to use a matching algorithm that emphasizes fairness. At the beginning of each round, the matching algorithm described in Figure 4 is performed. As can be seen in Figure 7, this leads to a much more equitable selection of slots for matching, with no harm to the rate of barriers per second. The slots which previously got less time slices because of their lower serial number, will now get an equal opportunity to be executed, so some of the jobs will pass more barriers per second, while other will pass less, but the total number of barriers which will be passed, is very similar to the one which the best-fit method had produced.

Sensitivity to Cluster Size

In the next set of experiments the measurements were repeated on clusters of sizes 2, 4, and 12 (in addition to the size of 8 used before).

Figure 8 shows the impact of the cluster's size on rate of progress of the different job types. Starting from the left, we see that when all the jobs are compute-bound the size does not have any real impact. But when compute-bound jobs are mixed with I/O-bound jobs, an interesting interaction occurs. As the processes of an I/O-bound job all perform I/O to the same server, a larger cluster size implies that more I/O operations are performed, leading to congestion at the server. Therefore the rate of progress of the I/O-bound jobs (measured in barriers per second, not total I/O done per second!) decreases when the cluster size is increased. This decrease would be avoided if the I/O itself was parallelized. But due to the use of paired gang scheduling, the reduced activity of the I/O-bound processes is picked up by the compute-bound processes. Thus the rate of progress of the compute-bound jobs improves with cluster size provided enough I/O-bound jobs

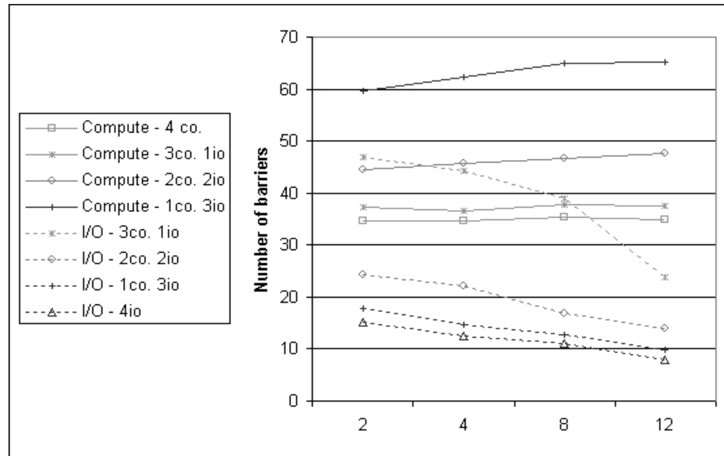

Figure 8: *Barriers per seconds in various size of clusters, for different mixes of compute and I/O jobs.*

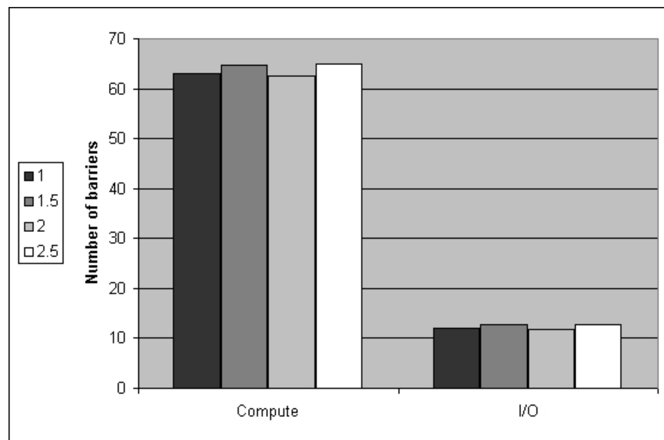

Figure 9: *Barriers per seconds in various granularities. The numbers in the legend are the size of the internal null loop in millions.*

are present.

Sensitivity to Granularity

In another set of these experiments, we changed the granularity of the compute part of the test's compute-bounded jobs. Specifically, the size of the internal null loop of the compute-bound jobs was reduced from 2.5 million down to 1 million, in steps of 500,000. This is an important test of the concept of paired gang scheduling, because there is a danger that when the granularity is finer, the asynchronous nature of the I/O-bound processes will cause more severe interference.

Figure 9 shows that in fact the different granularities do not have much impact on the rate of progress, despite the fact that the time between a barrier to the next barrier is smaller. Thus paired gang scheduling is not very sensitive to the granularity of the jobs.

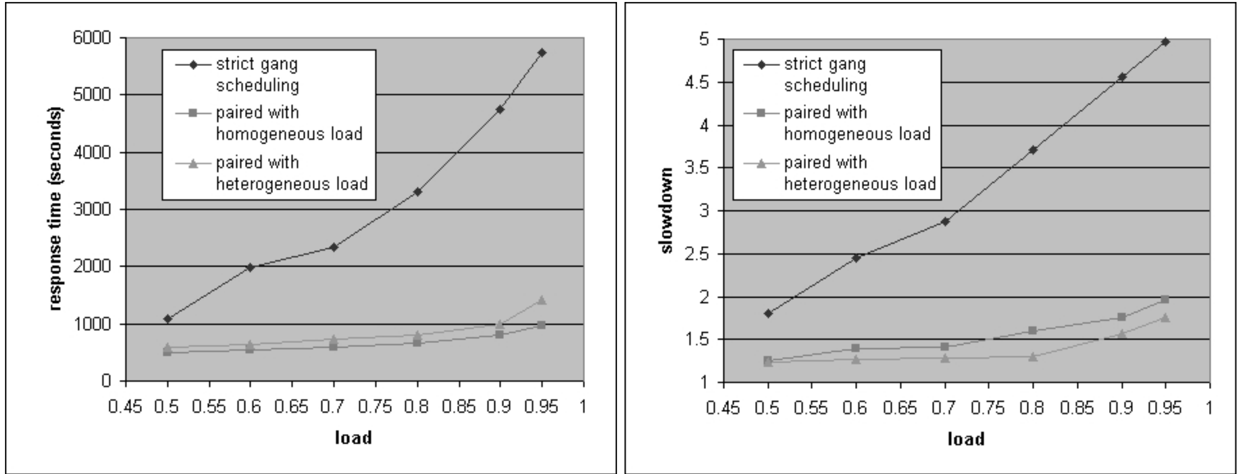

Figure 10: *Average response time and slowdown as function of system load.*

4.2 Performance with a Dynamic Workload

The results presented in the previous subsection were for a static workload containing a carefully controlled mix of jobs. While this allowed us to analyze the detailed behavior of the system, it was not very realistic. In this subsection we turn to more realistic workloads.

The experiments reported here were run on the full 16 nodes of the ParPar cluster. Each experiment consisted of running 1000 jobs generated according to the Lublin workload model [12]. The model specifies the arrival time, number of nodes, and running time of each job. As the model is based on long-running jobs from production supercomputer installations, we divided the arrival times and running times by 40 to reduce the overall time to run the experiments. The arrival times were further multiplied by a constant factor to achieve different load conditions. The time to complete a single experiment was typically in the range of 8–11 hours, with more time needed for the lower load conditions.

4.2.1 Homogeneous Fine-Grain Applications

In the first set of experiments all jobs are homogeneous, meaning that they display the same behavior. The selected behavior is one of fine-grain computation that still allows for matching. We started by measuring the time for the communication involved in a barrier synchronization, and setting the loop in the compute part to take slightly less time, so that the CPU utilization will be about 45%. We used the program described in Figure 6. N was set to $600 * sec$, Where sec is the number of seconds the job should run according to the workload model. G_{comp} was set to 25,000 and $G_{I/O}$ was set to 0. Since $G_{I/O}$ was 0, B is meaningless.

Our aim was to compare strict gang scheduling and paired gang scheduling. Exactly the same workload was used for all the different schedulers and load conditions. The results show that even under heavy loads, performance is reasonable when using paired gang scheduling. However, when using strict gang scheduling, the results show a saturation under heavy loads, created by a bottleneck of jobs, which the system cannot handle fast enough.

Figure 10 shows the average slowdown and response times of all the jobs with:

- strict gang scheduling.
- paired gang scheduling with a homogeneous workload.
- paired gang scheduling with a heterogeneous workload (described below).

The slowdown is calculated by $\frac{1}{N}\sum_{i=1}^{N}\frac{t_{Elapsed_i}}{t_{Executing_i}}$, where N is the number of jobs (in our case N is 1000), $t_{Executing_i}$ is the execution time of job i measured by the kernel on the nodes, and $t_{Elapsed_i}$ is the elapsed time of job i. The response time is simply the average of $t_{Elapsed_i}$.

Both response time and slowdown clearly show that paired gang scheduling can use the computer resources more efficiently. Hence, the jobs can complete their tasks in a shorter time period. When the workload is heavier, the ratio between paired gang scheduling and strict gang scheduling will be higher. Under 0.5 load, the response time with paired gang scheduling is about twice faster, while under 0.95 load the response time with paired gang scheduling is nearly 6 times faster.

Figure 11 shows the number of jobs in the system while running the simulation using these two methods of scheduling, for different load conditions. For low loads the strict gang scheduling and the paired gang scheduling are similar, and both schedulers give almost the same results. At all loads, the paired gang scheduling sometimes succeeds to clear all the jobs, while the strict gang scheduling fails to do so for loads above 0.6 or 0.7. At such high loads, after all 1000 jobs have been submitted, the scheduler still has some dozens of jobs left in the system. At the highest load of 0.95 it takes over three hours to successfully complete these jobs. In fact, for loads of 0.9 and 0.95 it seems that the system is actually saturated, as the number of jobs in the system seems to rise with time. This is due to wasting resources when processes wait for synchronization. Paired gang scheduling, on the other hand, does not saturate even at a load of 0.95.

Note that in these experiments we enable an unrestricted number of jobs in the system at the same time. This might be unrealistic for real applications, because of the memory pressure. However, this demonstrates that when paired gang scheduling is used, jobs can complete their computation faster. This leads to less overlap and a smaller degree of memory congestion, and the memory pressure is actually reduced significantly.

4.2.2 Heterogeneous Applications

The previous subsection focused on fine-grain applications, as they are expected to be the most sensitive to interference from other applications that share their time slot. To further increase the interference, we tailored the applications so that their CPU utilization would be roughly 45%, so that they will always be matched. The results indicate that even in this case the benefits of improved resource usage far outweigh the degradation due to interference.

The question we wish to tackle in this subsection is whether matching occurs enough in practice to make a difference. To do so, we need to create a mix of heterogeneous applications, with different granularities and CPU utilizations. As there is no real data about the mixes of CPU utilizations in production workloads, we will use a mix in which the CPU utilization is selected at random uniformly in the range from 0 to 100%, with the rest being used for barrier synchronizations. Thus, the granularities correlates with the utilization and for each degree of CPU utilization, we use the most fine-grain application possible.

The results are shown in Figure 10, and indicate that enough matching occurs to make the performance similar to that of the homogeneous case. Figure 12 shows the number of jobs in the system while running the heterogeneous load. As can be seen in this figure, there is just a

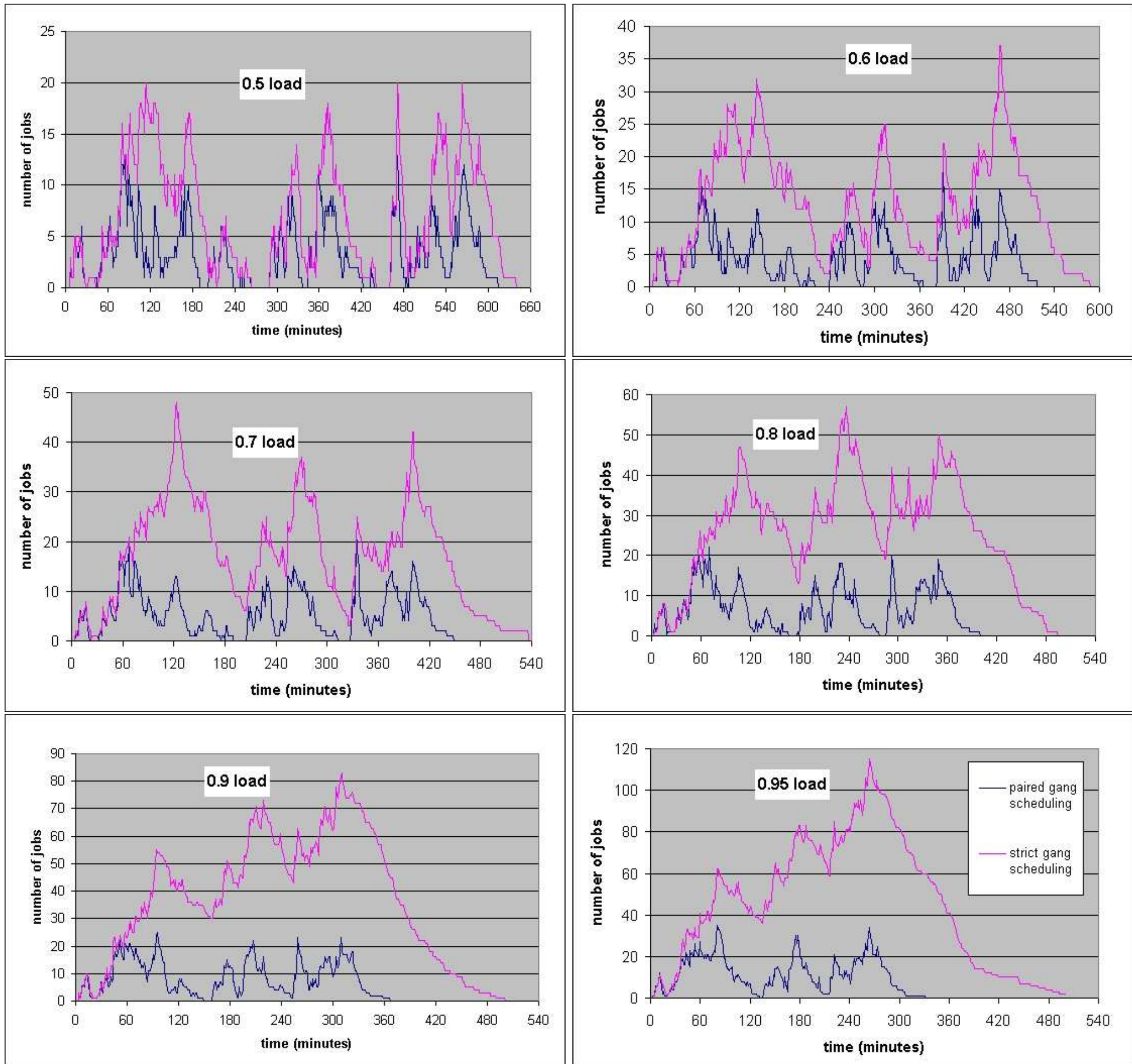

Figure 11: *Evolution of the simulation for different load conditions.*

slight difference between the homogeneous load and the heterogeneous load, when paired gang scheduling is used.

4.2.3 Applications with Distinct Phases

The applications used in the previous subsections were stationary, in the sense that their behavior did not change during the course of their execution. In particular, we used their phases to create fine-grain interactions, as we wanted to investigate how susceptible they were to interference from other applications. But real applications may have long phases of computation and I/O, and the pattern may also change during the run. In this section we investigate such applications, and check how paired gang scheduling adjusts to the changing behavior of applications.

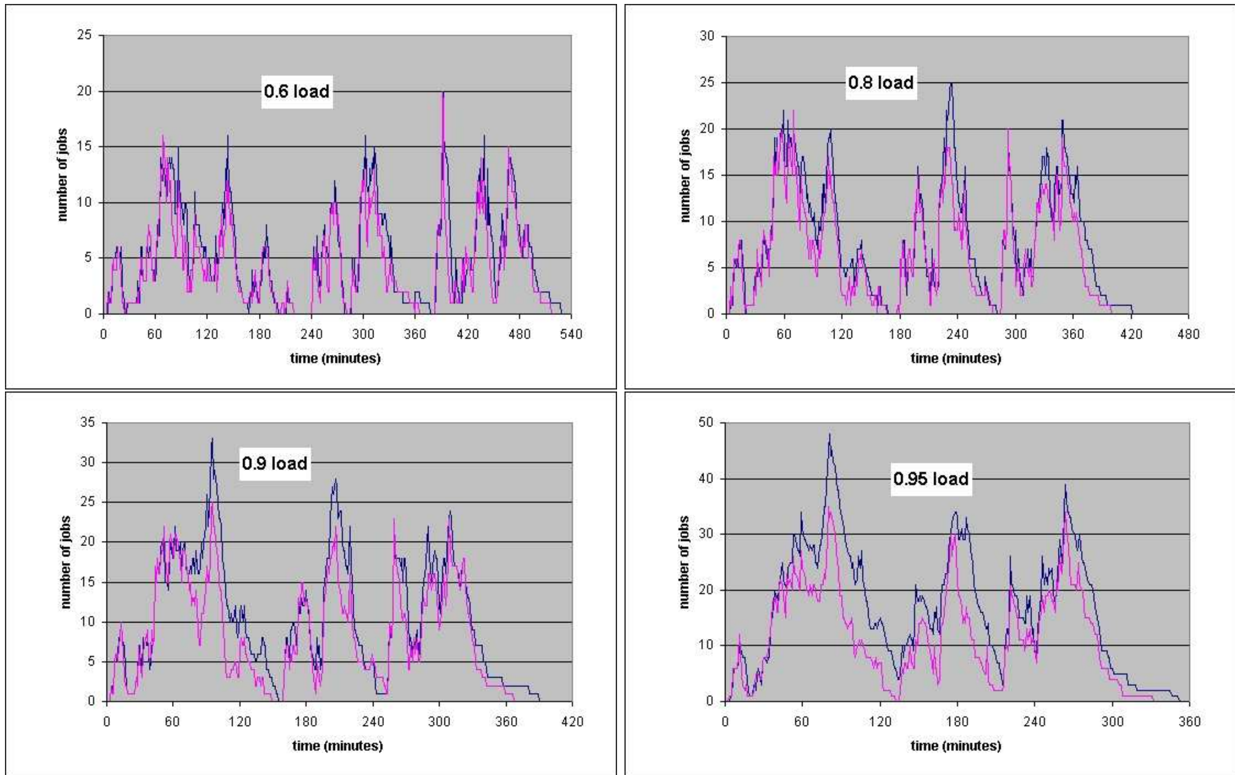

Figure 12: *Evolution of the simulation for heterogeneous loads.*

We used a model of the QCRD application as described in [15]. This application is structured as 12 short I/O-bound phases alternating with 12 longer CPU-bound phases. The final part of the application is a relatively long I/O-bound phase.

Figure 13 shows the CPU utilizations and the success of matching when 4 copies of QCRD are running. We can see that when the CPU utilization is high, just a single job runs at a time, while when the CPU utilization is low jobs are paired and run together.

First, the CPU-bound phase is done executing each job in a separate slot. Then the I/O-bound phase comes and the jobs are matched and run in pairs. When a new CPU-bound phase comes, it will take a little while for the scheduler to understand that there was a change and to run the CPU-bound phase alone. Sometimes, the scheduler executes a pair of CPU bound jobs, but it doesn't split them because it sees a CPU utilization lower than 50% for each one of the jobs. Note that this can only happen if the jobs are perfectly synchronized, which sometimes happens in this case where they are copies of the same application started together.

In order to solve the problem of the unnoticed changes by the scheduler, we changed the scheduler to detect any sharp change in the behavior of a job. When such a sharp change occurs, the scheduler will give the job an opportunity to run one quantum alone so its real characteristics can be observed. This might cause a little delay, because a pair of jobs which could be executed together are sometimes separated. However, it gives the scheduler a better way to know what is the real character of the jobs.

The results of the scheduler which notices sharp changes are shown in Figure 13. A sharp change was defined as a change of more than 20% in the CPU utilization. The response time was

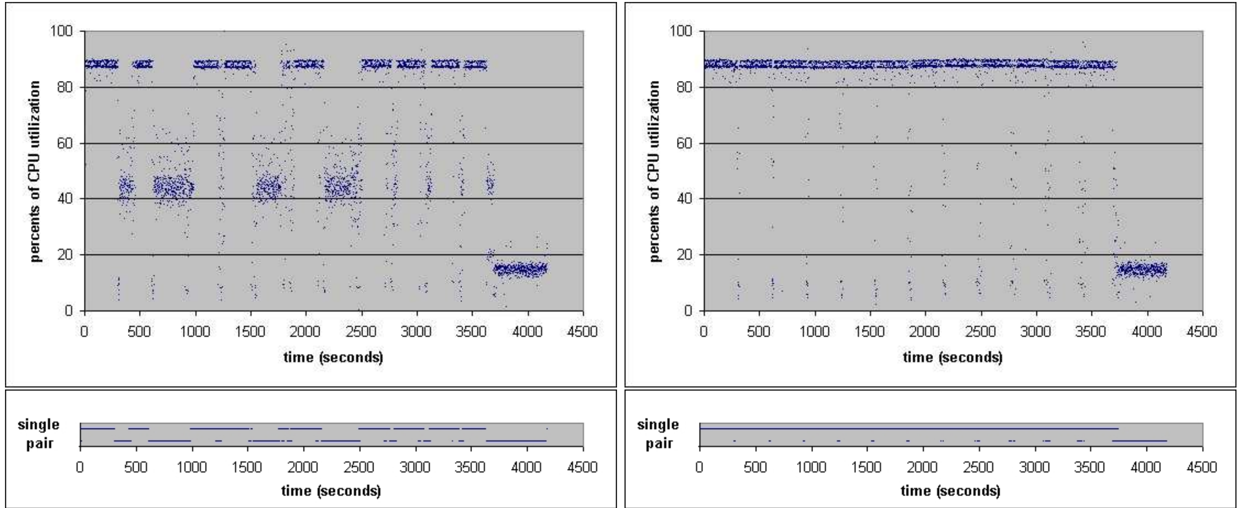

Figure 13: *Four instances of the QCRD model are executed in parallel. Left: original paired gang scheduling. Right: with special treatment of sharp changes.*

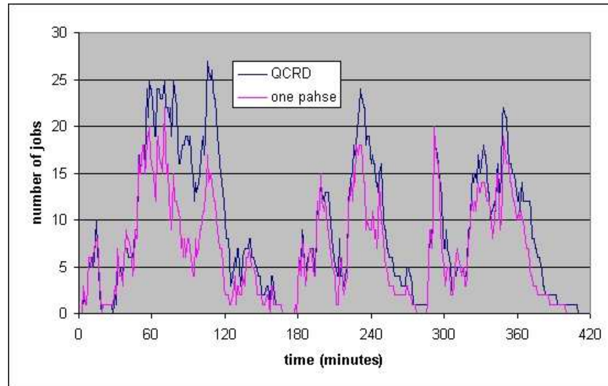

Figure 14: *A dynamic workload with QCRD-like jobs, load=0.8.*

longer by just about 1%.

We also experimented with a dynamic workload where all jobs are scaled versions of QCRD. Since the running time in the Lublin workload model is different for each job, we shrank the phases so the total time will be same as it is in the Lublin workload model. The results were similar to the results of the stationary applications, with only a very slight degradation. The progress of the runs can be seen in Figure 14, for a load of 0.8. When comparing the run of QCRD-like job under paired gang scheduling with running the same jobs using strict gang scheduling, the average response time was reduced by half.

5 Conclusions

The results of the experiments are promising: given a good match of a compute-bound parallel job and an I/O bound parallel job, they can run within the same quantum with little mutual interference, nearly doubling the resource utilization. Thus paired gang scheduling seems to be a good compro-

mise between the extreme alternatives of strict gang scheduling or uncoordinated local scheduling. Naturally, the system's ability to find such a match depends on the available job mix. Recent work has identified various I/O-intensive applications, so it seems that the potential is there [15, 20].

Especially noteworthy is the fact that this is achieved by a very simple device, based on data easily available directly to the scheduler. This distinguishes paired gang scheduling from other flexible gang scheduling schemes that are based on monitoring communication in order to deduce the characteristics of parallel jobs (e.g. [11, 7, 22, 1]).

The idea can be extended if there are more I/O devices in the system. One I/O operation won't interfere with other I/O operation, so a group of $N + 1$ gangs can be dispatched, where N is the number of I/O devices in the system. In particular, it is possible to overlap disk I/O with communication.

Additional future work includes the handling of user-level communication, in which poling and busy waiting may mask I/O activity and make it look like CPU activity. This requires modifications of the communications library, and in particular, the substitution of busy waiting by spin blocking.

Finally, we also intend to experiment with more advanced pairing schemes. For example, we would like to consider a more precise matching of slots, e.g. using a detailed measurement of CPU utilization on each processor rather than average values for each job.

References

[1] Arpaci-Dusseau A. C., Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Trans. on Computer Systems* **19(3)**, pp. 283–331, August 2001.

[2] Etsion Y. and Feitelson D. G., User-level communication in a system with gang scheduling. *15th Intl. Parallel & Distributed Processing Symp.*, Apr 2001.

[3] Feitelson D. G., Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS Vol. 1162, pp. 89–110, 1996.

[4] Feitelson D. G., Batat A., Benhanokh G., Er-El D., Etsion Y., Kavas A., Klainer T., Lublin U., and Volovic M. A., The ParPar system: a software MPP. In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, Rajkumar Buyya (Ed.), pp. 754–770, Prentice-Hall, 1999.

[5] Feitelson D. G. and Jette M. A., Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, Springer Verlag, LNCS Vol. 1291, pp. 238–261, 1997.

[6] Feitelson D. G. and Rudolph L., Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, 1992.

[7] Frachtenberg E., Feitelson D. G., Petrini F. and Fernandez J., Flexible CoScheduling: mitigating load imbalance and improving utilization of heterogeneous resources, *17th Intl. Parallel & Distributed Processing Symp.*, Apr 2003.

[8] Franke H., Pattnaik P., and Rudolph L., Gang scheduling for highly efficient distributed multiprocessor systems. *6th Symp. Frontiers Massively Parallel Comput.*, pp. 1–9, Oct 1996.

[9] Hori A., Tezuka H., Ishikawa Y., Soda N., Konaka H., and Maeda M., Implementation of gang-scheduling on workstation cluster. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS Vol. 1162, pp. 126–139, 1996.

[10] Kavas A., Er-El D., and Feitelson D. G., Using multicast to pre-load jobs on the ParPar cluster. *Parallel Comput.* **27(3)**, pp. 315–327, Feb 2001.

[11] Lee W., Frank M., Lee V., Mackenzie K. and Rudolph L., Implications of I/O for gang scheduled workloads. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS vol. 1291, pp. 215–237, 1997.

[12] Lublin U. and Feitelson D. G., *The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs*. Technical Report 2001-12, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Oct 2001.

[13] Mu'alem A. W. and Feitelson D. G., Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel & Distributed Syst* **12(6)**, pp. 529-543, June 2001.

[14] Ousterhout J. K., Scheduling techniques for concurrent systems. *3rd Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[15] Rosti E., Serazzi G., Smirni E., and Squillante M. S., Models of parallel applications with large computation and I/O requirements. *IEEE Trans. on Software Engineering* **28(3)**, pp. 286–307, Mar 2002.

[16] Rosti E., Serazzi G., Smirni E. and Squillante M. S., The impact of I/O on program behavior and parallel scheduling. *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.* pp. 56–65, 1998.

[17] Schwiegelshohn U. and Yahyapour R., Improving first-come-first-serve job scheduling by gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS Vol. 1459, pp. 180–198, 1998.

[18] Silberschatz A. and Galvin P., *Operating System Concepts*. 5th ed., Addison-Wesley, pp. 130–133, 1997.

[19] Silva F. A. B. and Scherson I. D., Improving parallel job scheduling using runtime measurements. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS vol. 1911, pp. 18–38, 2000.

[20] Smirni E. and Reed D. A., Workload characterization of input/output intensive parallel applications. *9th Intl. Conf. Comput. Performance Evaluation*, Springer-Verlag, LNCS vol. 1245, pp. 169–180, Jun 1997.

[21] Snavely A., Tullsen D. M. and Voelker G., Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.* pp. 66–76, Jun 2002.

[22] Sobalvarro P. G., Pakin S.,Weihl W. E. and Chien A. A., Dynamic coscheduling on workstation clusters. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, LNCS vol. 1459, pp. 231–256, 1998.

[23] Wiseman Y. and Feitelson D. G., Paired gang scheduling. *Jerusalem Parallel & Distributed Processing Symp.*, Jerusalem, Israel, Nov 2001.

[24] Zhang Y., Franke H., Moreira J. and Sivasubramaniam A., Improving parallel job scheduling by combining gang scheduling and backfilling techniques. *Intl. Parallel & Distributed Processing Symp.*, pp. 133–142, May 2000.