Job Scheduling with Lookahead Group Matchmaking for Time/Space Sharing on Multi-core Parallel Machines

Xijie Zeng and Angela Sodan

University of Windsor, Windsor ON N9B 3P4, Canada
zengx@uwindsor.ca,acsodan@uwindsor.ca

Abstract. Multi-core nodes of parallel machines may only provide gradual performance improvement per application due to competition on resources like the cache. As shown in our earlier work, spreading out applications over as many nodes as possible or letting different applications with potentially complementary characteristics (semi time) share each node by allocating different cores to them may provide better performance. In the latter case, groups of jobs may be necessary to obtain balanced resource utilization due to different sizes of jobs. We present a scheduler G-LOMARC-TS which can match groups of jobs and consider both space- and time-sharing allocation. Since matchmaking may select jobs further down in the waiting queue, fairness in regards to possible delays of the other jobs is watched and delays are kept within certain bounds. This results in a large number of possible combinations. A number of heuristics to select the most promising combinations make it possible to deal with the NP-completeness of the problem. We show that our scheduler improves utilization of high-load phases by about 27% and subsequently average response times by about 36% (and 53% for long jobs) compared to space sharing scheduling for normal workloads. Additionally the scheduler can handle much higher workloads than a space-sharing scheduler.

Keywords: space sharing, semi time sharing, lookahead matchmaking, job groups

1 Introduction

Multi-core nodes in cluster are becoming widespread though the additional cores may only provide gradual performance improvement due to competition on shared resources such as memory, network, and potentially caches. In earlier work, we have shown that better results may be obtained if rather using the additional cores for other applications with complementary characteristics [14][16][20], as also found by other researchers [18]. We call such resource allocation *semi time sharing* since the cores are partitioned among the applications, i.e. space-shared, but the other resources may be shared.

Our LOMARC scheduler [16] first proposed such semi time sharing on nodes with hyperthreaded CPUs, assuming that the second virtual CPU per CPU/node

would either be allocated to a second application or remain unused. We proved that many combinations of the NAS benchmarks ran very well together. Under the conditions mentioned, LOMARC improved average response times by 30% to 50%. In other work, we showed that even if the choice is between using fewer nodes exclusively (space sharing) vs. more nodes coscheduled with another application (semi time sharing), semi time sharing may perform better [15]. We also found that pairing communication-bound applications may yield acceptable slowdown [15][20]. While our work focused on computation and communication performance, the study in [18] focused on memory and I/O, confirming the benefits from coscheduling. However, there are also cases where an application can exploit the performance potential of all cores per node well and space sharing may perform better.

To use the cores per node most effectively, more general schedulers are needed that can allocate the cores intelligently for high resource utilization. Similar to adaptive approaches which reduce job sizes under high load [13], we can expect that higher resource utilization decreases average response times in spite of increasing individual runtimes, because load and subsequently average wait times are significantly decreased.

The LOMARC scheduler [16] applied lookahead matchmaking among waiting and running jobs to find jobs with complementary usage of multiple resources, using simple heuristics to select the best combination. However, no fairness considerations were applied in regards to the impacts of the partial reordering of the waiting queue.

Our main goals for the work presented in this paper are to

- Provide a more flexible approach for nodes with multiple cores and for applications which may either exploit multiple cores per node exclusively or share them with another application. We can consider this approach as semi adaptive by assuming a fixed number of processes that can be allocated differently to nodes and cores.
- Provide a framework which, in spite of partial reordering of the waiting queue that is necessary to find suitable matches, is fair to individual jobs and does not impose overly high delays on individual jobs.
- Match groups of jobs since jobs typically have very different job sizes and only matching two jobs may therefore not provide sufficient potential for utilization gain.

The main contributions of this paper and our new G-LOMARC-TS scheduler are:

- Matching groups of jobs among waiting and/or running jobs.
- Applying several heuristics that are likely to extract the most promising groups since finding the optimum group among the many possibilities is an NP-complete problem.
- Including important special cases like bursts of serial jobs.
- Using a metric for selection of the best group which considers the absolute gain in utilization (nodes saved over a certain time interval) and subsequently the global interest of all jobs.

– Supporting both space and time sharing and choosing between them according to the utilization gain and current machine load.
– Providing fairness to individual jobs by using a maximum slack factor vs. the originally estimated response time as a constraint for the reordering.
– Providing more chances to fit a job into the machine by including the options of 1) matching it with running jobs and 2) reducing its node requirement via space sharing.

G-LOMARC-TS is implemented as an extension of the coarse-grain preemption scheduler Scojo-PECT [5]. We demonstrate via simulation with synthetic workloads that G-LOMARC-TS performs significantly better than pure space sharing and that group matching contributes significantly to the improvements and is therefore essential.

The paper discusses related work in Section 2. Section 3 defines the machine and application model. Our G-LOMARC-TS algorithm is described in Section 4, including utilization-gain and slowdown metrics. Experimental results are shown in Section 5, and Section 6 gives a summary and conclusion.

2 Related Work

Existing approaches to time sharing for parallel jobs are gang scheduling [5] and loosely coordinated coscheduling [13]. Gang scheduling allocates globally synchronized time slices, while keeping all jobs in memory to make slice switches fast. Loosely coordinated coscheduling uses distributed algorithms to approximate coordinated execution which is necessary to avoid idling in communication. Though gang scheduling may better pack jobs into the machine, it does not improve utilization of the individual resources such as network or disk. Loosely coordinated coscheduling has the potential of improving utilization by switching between jobs to hide resource-access latencies. However, jobs need to be fairly synchronous or very coarse-grain to make coordinated execution possible or unimportant, respectively. Proposals were made to relax gang scheduling and merge time slices with computation-bound and I/O-bound jobs [19] or to switch from gang scheduling to loosely coordinated coscheduling for coarse-grain or I/O-bound jobs [1]. Both approaches depend on the dynamic availability of suitable job combinations but can adapt to different phases in the program execution. However, the probability of finding suitable dynamic job combinations decreases if groups of jobs need to be formed.

In regards to semi time sharing, studies found complementary characteristics of the coscheduled jobs to perform better due to balancing the usage of system resources [16][18]. Spreading out jobs to different nodes and semi time sharing the resources per node among multiple jobs may perform better than dedicated allocation of all node resources to one job [14][15][18]. The experiments in [18] were carried out with only 4 nodes and therefore limited communication but the experiments in [14][15] on up to 64 nodes show that the benefits of semi time sharing vs. dedicated allocation scale to larger number of nodes. Characteristics

which were found important for job combinations are whether the job is CPU-bound, cache-bound, memory-bound, network-bound, or disk-bound [16][18] but also which access patterns are applied [10][20]. For example, CPU-bound jobs were shown to match well with network-bound or memory-bound jobs. Several studies on hyperthreaded and multi-core CPUs showed that scheduling multiple processes of the same job per node provides limited benefits or does not work well in certain cases. Thus, in [2], jobs became only between 1.2 and 1.5 times faster by running another process on a second AMD-Opteron core. Potentially high resource contention on hyperthreaded CPUs from processes of the same application with same resource requirement (such as the cache) were shown before in [7][8]. In other cases, resource sharing per node can provide a benefit rather than performance degradation: intra-node communication through shared memory and cache may be faster than inter-node communication which is a benefit if a large percentage of messages are transferred via intra-node communication [4] (50% of the messages were found to be intra-node).

Thus, the LOMARC scheduler [16] matches jobs with complementary characteristics at job start times, while partially reordering the waiting queue to find suitable matches. The work in [18] proposes a scheduler which space-partitions each node into one half for memory-bound and one half for the other jobs.

Fairness is discussed in several papers. The work in [11] measures overall fairness (in retro) of a job scheduler by considering the actual start time of a job vs. its virtual start time without effects from later arriving jobs. The slack approach in [17] tries to maintain relative fairness among jobs by dynamically calculating possible delays in the presence of different job priorities.

3 Machine and Application Model

We assume that the target machine is a cluster with multiple multi-core CPUs per node. Though multiple CPUs and multiple cores per node can significantly increase performance, they do not simply multiply the performance by the number of CPUs/cores due to the contention effects on shared resources, but rather typically provide less performance gain than additional nodes. Processes running on the same node share the network, disk, and the memory. Processes running on the same CPU additionally share the memory access paths and potentially the cache. In regards to the cache, some multi-core CPUs share the L2 cache (such as the Intel Core Duo, IBM POWER5, and Ultra SPARC T1/T2), whereas other multi-core CPUs have private L2 caches per core (AMD Opteron, Intel Itanium, IBM POWER6, and Ultra SPARC IV). We model the contention effects as application slowdown (Section 4.9), and differentiate between CPU and core slowdowns, with CPU slowdowns typically being lower. This also implies that the allocation to CPUs and cores matters if fewer processes run per node than there are CPUs/cores available. In the few cases where resource sharing among processes of the same application provides a benefit, the slowdown would turn into a speedup.

We assume that jobs and workload have the following characteristics:

– Jobs consist exclusively of processes (no threads) which is still the dominant approach applied by users [2].
– Upon job submission, the number of processes (the job size) is specified but the allocation to nodes, CPUs, and cores is left to the scheduler. The number of processes is therefore fixed (no molding in the sense of changing the job size).
– Runtime estimation and sufficient characteristics information to calculate slowdowns are available, provided by users, historical databases, or compilers. General research progress made by compilers, application profilers, and by prediction from historical information suggests that roughly correct runtime estimation by the system is becoming realistic for future schedulers (see e.g. [3]). More optimistic is the assumption about slowdown estimations being available which is farer in the future. This assumption helps to explore possible benefits obtainable from such information and its exploitation in advanced space/time-sharing job schedulers.
– The workload includes a large percentage of serial jobs and of parallel jobs with power-of-two sizes, as observed in the analysis of job traces [9]. Also bursts of submissions (submission of jobs with potentially similar characteristics in close time proximity) are possible.

Table 1. Terms used throughout formulas in paper.

Term	Meaning
S_i	Size (all processes) of Job i
T_i	Runtime of Job i if scheduled individually with 1 process per node
PPN_i	Number of processes of Job i per node
$T_{makespan}$	Total runtime of workload
M	Machine size in number of nodes
N_{core}	Number of cores per node
U_{core}	Core utilization
U_{node}	Node utlization
U_{gain}	Utilization gain for comparison of node usage
$R_{est,i}$	Estimated response time of Job i in FCFS order at submission time
R_i	Response time of Job i
F_{slack}	Slack factor used for fairness check

4 G-LOMARC-TS Scheduling Algorithm

4.1 Scheduling Objectives

Terms used in the following discussion of formulas are listed and explained in Table 1.

The objective of the scheduler is to obtain best possible utilization in phases of high load, while keeping fairness acceptable. Higher utilization in high-load phases likely leads to better average response times [12]. Note, that utilization remains equal to the submitted load, independent of the scheduling policy, as long as the scheduler is not saturated (i.e. can handle the offered load and jobs do not queue up). Schedulers with little utilization support would delay jobs until phases with lower load, whereas the machine may be idle or very lowly loaded in such phases for schedulers with high utilization support.

Thus, we use utilization improvement in phases of high load as the primary objective and fairness as a constraint. This requires to formally define high-load phases, core and node utilization, utilization gain, fairness, and the scheduler impact on fairness–which we do in the following.

To formalize utilization, we assume that N_{wait} is the number of jobs in the waiting queue, N_H is a threshold for rating as high load, tp_i is a time period between load changes (due to job termination, job start, and slice switches), and t_l, t_j, and t_k are certain time points of load changes, un_i and uc_i are the number of used nodes and used cores, respectively, during the time period tp_i. High-load phases $Phase_H$ are defined as:

$$\begin{aligned} Phase_{H,l} = [t_l \text{ with } N_{wait} \geq N_H \;\&\&\; N_{wait} < N_H \text{ for } t_{l-1}, t_j \text{ with } N_{wait} < N_H \\ \&\& \text{ any } t_k \text{between l and} j \text{ has } N_{wait} \geq N_H] \end{aligned} \tag{1}$$

Node utilization U_{node}, is the percentage of used-nodes time over the makespan.

$$U_{node} = \sum_{i \text{ in timeperiods}} (tp_i * un_i)/(T_{makespan} * M) \tag{2}$$

Core utilization U_{core}, is the percentage of used-cores time over the makespan.

$$U_{core} = \sum_{i \text{ in time periods}} (tp_i * uc_i)/(T_{makespan} * M * N_{core}) \tag{3}$$

Similarly, U_{node} during a high-load phase is the percentage of used-nodes time over this phase and U_{core} is the percentage of used-cores time over this phase.

Utilization gain U_{gain} (for details, see Section 4.6) is measured in saved usage of nodes and used as the metric per individual scheduling decision. Nodes are saved via increased core utilization–which, as mentioned above–is only a meaningful overall metric if confined to phases of high load but always makes sense per individual scheduling decision.

In regards to fairness, there exist different definitions of fairness in the literature. We consider any delay versus the response time without reordering and coscheduling as a negative impact on fairness. Since our scheduling is basically FCFS with conservative backfilling, estimation of response times via simulation at submission time is possible, and we record the estimated response times ($R_{est,i}$). Fairness is then provided by limiting the response-time changes

to $R_{max,i}$ (response time can increase due to the matchmaking reordering), calculated via slack factor (F_{slack}):

$$R_{max,i} = F_{slack} * R_{est,i} \tag{4}$$

Finally, we define our objective as maximizing core utilization U_{core} during high-load phases $Phase_H$, while maintaining $R_i \leq R_{max,i}$. The optimization is approximated by heuristics, making per-job-group decisions, and calculating utilization gain per decision.

Figure 1 shows how high core utilization leads to saved nodes usage if 6 jobs are scheduled by our G-LOMARC-TS compared to space sharing (note that for Job 6, number of processes per job per node (PPN) is 2 with G-LOMARC-TS and 4 with space sharing; i.e, the node requirements are different under the two scheduling schemes).

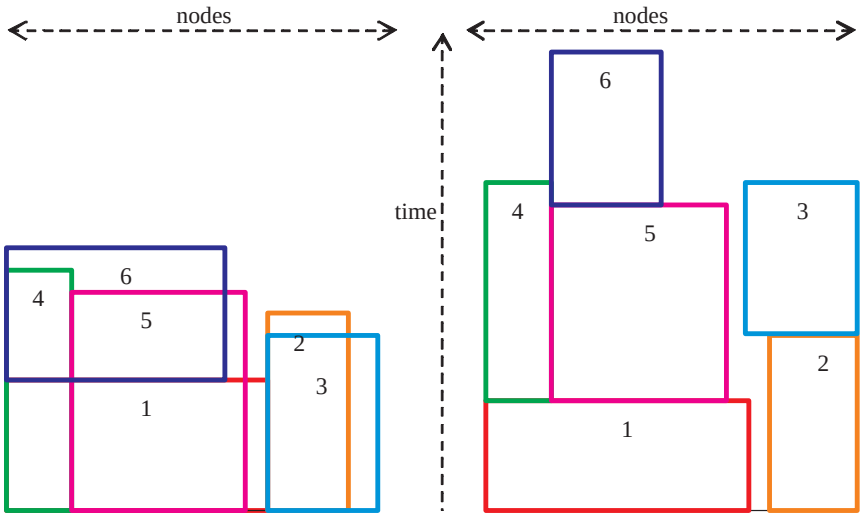


Fig. 1. Utilization gain with G-LOMARC-TS from increased core utilization for Job 1 to Job 6 (left) compared to space sharing (right).

4.2 General G-LOMARC-TS Scheduling Idea

Our G-LOMARC-TS supports both space sharing and semi time sharing which we define more precisely as follows:

– *Space sharing*: Resources are allocated in a dedicated manner, but the machine (its "space") is shared, i.e., different parallel jobs may potentially run

at the same time if resource requirements permit them to be allocated to different subsets of compute nodes.

– *Semi time sharing*: The CPUs/cores per SMP node of a cluster are allocated to different jobs as illustrated in Figure 2. Semi time sharing does not share and switch the core as done under standard time sharing. However, other resources like memory, network, disk and potentially caches (Section 3) are simultaneously shared. Since all cores remain responsive to communication, no coordination of parallel processes is required as necessary under standard time sharing.

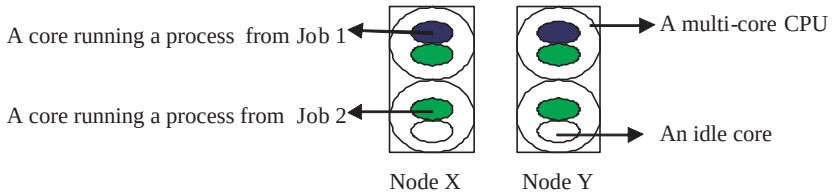


Fig. 2. Coscheduling processes of different parallel jobs per node.

Processes of the same job have similar characteristics, contention on certain resources is very likely for these processes if they share resources on the same node. However, different jobs may have different characteristics and complementary resource usage, leading to less contention. Thus, our G-LOMARC-TS scheduler supports the option of coscheduling, i.e. scheduling processes from different applications on the same nodes, see Figure 2. To make such coscheduling effective, job combinations with high complementary resource usage should be created. This does typically not apply if only pairing the first 2 jobs in the waiting queue. Rather the scheduler needs to search among waiting and running jobs for suitable matches. This means lookahead in the waiting queue vs. FCFS order. If jobs move ahead by being coscheduled with the first job in the waiting queue, typically the runtimes increase (due to contention). This delays the first and other jobs originally in front of the coscheduled ones in the queue. Runtimes also increase for running jobs if waiting jobs are coscheduled with them. Thus, the matchmaking leads to partial reordering of the waiting queue and potential delays for running and waiting jobs. This is the reason why we need to include the fairness criterion as described in Section 4.1 to avoid severe delays or push-backs for individual jobs.

However, in some cases the processes of the same job may run well together and may even benefit from intra-node communication (Section 3). In such cases, space sharing is the better option. Thus, our scheduler supports both coscheduling and individual scheduling.

In regards to coscheduling, the simplest approach is pairing two jobs as applied in the original LOMARC scheduler. However, sizes of jobs can be very different. Therefore grouping multiple jobs for coscheduling can increase the chance for utilization gain. We have the following cases of forming groups: 1) matching a waiting job with several other waiting jobs, 2) matching a waiting job with several running jobs, and 3) matching a running job with several waiting jobs.

4.3 Time vs. Space Scheduling

In the following, we explain the space sharing and semi time sharing options of G-LOMARC-TS in more detail. Space and semi time sharing can involve different numbers of processes of the same job per node if we have more than 2 cores per node. For example, with 4 cores per node, we can have 1 or 2 processes per job per node with semi time sharing, and we can have 1, 2, or 4 processes per node per job with space sharing. Which number is chosen depends on the self slowdown caused by resource contention of processes of the same job (details are discussed in Section 4.9). If the self slowdown is severe, fewer processes per node per job are meaningful. If the self slowdown is low or if there is even speedup, more processes per node per job are better. For space sharing, the number of processes per node is always chosen to utilize the space well, balancing runtime with used cores by employing a threshold on acceptable self slowdown.

In our scheduler, short jobs are only scheduled via space sharing because short jobs are not considered worth the effort of matchmaking. Otherwise, decisions between space and semi time sharing are made adaptively when trying to schedule the first job in the waiting queue. If the workload is low in the sense that all jobs fit into the machine with one process per node, space sharing is more beneficial because obtaining the best runtimes per job. A special case is that there may not be enough space to schedule a job under space sharing but it may be possible to start the job by coscheduling it with running jobs. Otherwise, whether space or semi time sharing is applied depends on which option provides better utilization gain for the current scheduling decision.

If a job's self slowdown is low, it may benefit more with space sharing where only self slowdown involves. However, if a job can find matched jobs with complementary resource requirements and consequently with low coscheduling slowdown caused by processes of different jobs (described in Section 4.9), it may gain more with semi time sharing.

4.4 Scheduling Algorithm

The main part of the G-LOMARC-TS scheduling algorithm is shown in Figure 3. The algorithm description is generalized to work with any number of cores per node (though our evaluation uses 4 cores per node). The scheduler tries different scheduling possibilities: space sharing and semi time sharing matching the first waiting job with other waiting jobs or running jobs. Space sharing is used if the load is low. Serial jobs are treated specially as described in Section 4.5, before attempting other forms of coscheduling. Then, the 3 forms of semi time sharing

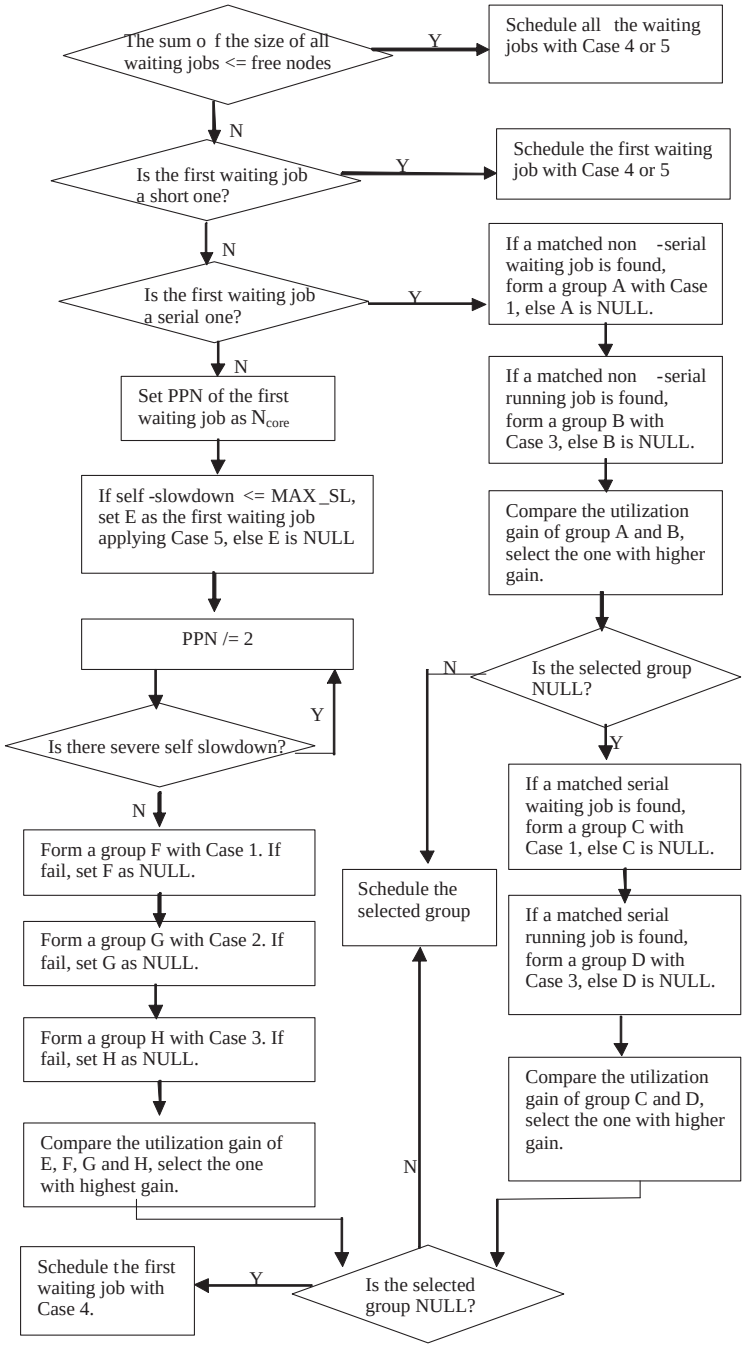


Fig. 3. Flow chart for main part of scheduling algorithm.

(Case 1 to Case 3) and space sharing with full usage of all cores per node (Case 5) are compared. The option with the highest utilization gain is selected. If all of the latter options fail, space sharing with partial usage of the cores per node (Case 4) is applied.

For coscheduling, groups are formed. A *group* is composed of a *primary job* and one or multiple *matched jobs*. The primary job can be the first waiting job or a running job. (Details about forming groups are described in 4.5.)

Then, the first waiting job can be scheduled/coscheduled in the following ways:

- Case 1 (semi time sharing): coscheduled in a group with the first waiting job as the primary job and several other waiting jobs as matched jobs (one waiting : multiple waiting).
- Case 2 (semi time sharing): coscheduled in a group with the first waiting job as the primary job and several running jobs as matched jobs (one waiting : multiple running).
- Case 3 (semi time sharing): coscheduled in a group with a running job as the primary job and several waiting jobs including the first one as matched jobs (one running : multiple waiting).
- Case 4 (space sharing): scheduled individually with $PPN = 1$ or 2 or 4 ... or $M - 2$
- Case 5 (space sharing): scheduled individually with $PPN = M$.

A similar algorithm is applied when attempting to backfill jobs. However, only Case 1, Case 4, and Case 5 are applied.

Note that groups of jobs are scheduled as a whole and adding individual jobs later is not considered. Nevertheless, a job may be matched more than once over its runtime since the group may be disbanded and the job become an individual job again. Disbandment of a group happens under the following conditions:

- The primary job terminates, while at least one matched job is still running.
- All matched jobs terminate, while the primary job is still running.

This also means that the scheduler makes no attempt to add jobs to a running group if some of the matched jobs terminate.

4.5 Group Formation

Forming groups is an NP-complete problem due to the many possibilities to combine jobs with different runtimes and sizes and due to slowdowns depending on the job combination. To make the problem tractable, we apply intelligent heuristics to form groups.

As mentioned above, a group is composed of a primary job and one or multiple matched jobs which may be waiting or running jobs. Per node, only two jobs are coscheduled (one is the primary job, the other one is one of the matched jobs). If the primary job is a running job, we choose of the least delayed jobs (since coscheduling implies slowdown).

After the primary job has been decided, the matched jobs are selected with the following steps (if the primary is the first waiting job, the matched jobs are running or other waiting jobs; if the primary is a running job, the matched jobs are waiting jobs):

1. Pre-selection: If a job and the primary job do not slow down each other severely (less than a threshold), the job is selected as a candidate for matched jobs.
2. Sorting: Candidate jobs are sorted in increasing order of delay if they are running jobs; if they are waiting jobs, their original FCFS order is kept. Then the sorted candidate jobs are divided into blocks, and the jobs per block are sorted in decreasing order of their remaining runtime.[1]
3. First block: A "window" with the same node requirements as the primary job "slides" over the first block (see Figure 4). Each time, the set of jobs within the window's range is selected as matched jobs (though we permit the aggregated node requirements of the matched jobs to be slightly larger than the window). The set with the highest utilization gain achieved is selected as the best group. Likely, most of the matched jobs in the best group are from the first block if including the fairness constraint.
4. Other blocks: If the primary job is not coscheduled over all its nodes (there is still space left) in the best group, jobs from the remaining blocks may be added if this leads to an increase in utilization gain. Each new group which increases the utilization gain is stored.
5. Purify: Matched jobs which slow down the primary job most but do not contribute to the utilization gain are removed from the group.
6. Fairness check: A group which causes any other job to be delayed severely (more than R_{max}) is discarded. Groups kept in Step 4 are tested for fairness in decreasing order of utilization gain until a group passes the check.

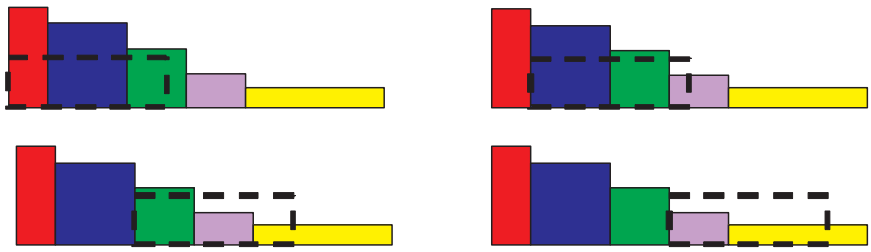


Fig. 4. "Window" with shape of primary job sliding over first block to search for a set of matched jobs.

[1] The order could also be chosen as least delayed vs. their estimated response time but experimental results show virtually no difference.

4.6 Utilization-Gain Calculation

If the cores per node are better utilized (more processes running per node, fewer idle cores), fewer nodes will be used to run a specific number of jobs. Thus, as discussed in Section 4.1, this means that high core utilization leads to saved node usage. Utilization gain is calculated as the ratio of saved nodes to used nodes which we first explain on the basis of the example shown in Figure 5. Figure 5 (left) shows the resource requirements of Job 1, 2 and 3 when no coscheduling scheme is applied, i.e. the runtime does not have any slowdown. Figure 5 (right) shows a group formed by these three jobs with Job 3 as the primary job and Job 1 and 2 as matched jobs. In this group, each job has two processes per node, i.e., the node requirement is half of its process number and the runtime is extended by the slowdown. After T_g time, Job 3 in the group finishes running and the group is consequently disbanded. All the work of Job 2 and 3 is done during the group execution and part of the work of Job 1 is done. Let us assume that, in order to do the same amount of work, Job 1 has to spend T_1 time without any coscheduling scheme, Job 2 spends T_2, and Job 3 spends T_3. We compare the node usage of the group to the sum of the node usages of the three jobs without coscheduling and calculate the utilization gain U_{gain} of the group as

$$\begin{aligned} U_{gain} &= (T_1 * S_1 * N_{cores} + T_2 * S_2 * N_{cores} + T_3 * S_3 * N_{cores} - \\ &\quad T_g * S_3 * N_{cores}/2) \ / \ (T_g * S_3 * N_{cores}/2) \\ &= (T_1 * S_1 + T_2 * S_2 + T_3 * S_3 - T_g * S_3/2)/(T_g * S_3/2) \end{aligned} \tag{5}$$

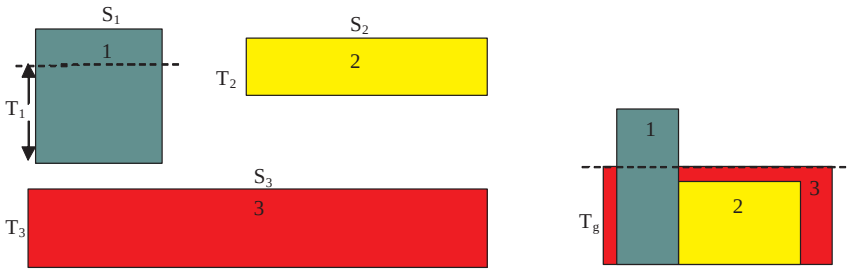


Fig. 5. Node usage of Job 1 to Job 3 with space sharing (left) and semi time sharing, i.e. coscheduled (right).

To generalize the calculation for all groups, we assume that a primary Job A coschedules with N matched jobs (Job 1, 2, 3, ... Job n). Suppose that $SL_{A,i}$ is the slowdown of Job A when A coschedules with Job i; $SL_{i,A}$ is the slowdown of Job i when it coschedules with A. There are Q time periods t_q $(1 \leq q \leq Q)$ during the coscheduling of the jobs between changes in the coscheduling status

(jobs terminating). $Q \leq N$ because the group is disbanded if all matched jobs or the primary job have terminated. To make the formula easier to understand, and without loss any generalization, we assume that the sum of the node requirements of all matched jobs is less than or equal to that of the primary job (the real algorithm permits that the node requirement of the matched jobs is slightly greater than the primary job).

In each time period q, there are F_q matched jobs running and the indexes of the matched jobs are x_1, x_2, x_3, ..., x_{F_q}. SL_q $(1 \leq q \leq Q)$ is the maximum slowdown of Job A when A coschedules with the matched F_q jobs running in time period q. Then, $SL_q = max\{SL_{A,x_1}, SL_{A,x_2}, \ldots, SL_{A,x_{F_q}}\}$ and the sum of all time periods t_{total} is $t_{total} = \sum_{1 \leq q \leq Q} t_q$.

Then, the general formula for calculating the utilization gain is:

$$U_{gain} = (\sum_{1 \leq q \leq Q} (\sum_{i \; in [x_1, x_2, \ldots, x_{F_q}]} (t_q * S_i / SL_{i,A}) + t_q * S_A / SL_q) - t_{total} * S_A / PPN_A) \; / \; (t_{total} * S_A / PPN_A) \quad (6)$$

Note that utilization gain is defined as a relative gain. This means that matching shorter jobs is given preference if other parameters are the same. Experimental results show that this gives a minor benefit vs. using absolute gain though it does not make any substantial difference.

4.7 Incorporation into Scojo-PECT

Scojo-PECT [5] employs preemption to support scheduling of shorter jobs even in the presence of long-running. Scojo-PECT preempts jobs to swap space which is easy to support in the machine environment and avoids the memory pressure which gang scheduling imposes. Scojo-PECT does not require hard-to-support checkpointing but subsequently imposes the constraint that preempted jobs are later restarted on the same resources as migration is not possible without checkpointing [13]. To make preemption to disk affordable and avoid that jobs are delayed because of problems to get access to their resources again, Scojo-PECT employs coarse-grain time slices and preempts all jobs. Jobs are sorted according to job type based on their runtime (we currently support short, medium, and long jobs), and scheduled in different virtual machines with a time slice per job type / virtual machine. The slice time for each job type is determined on the basis of typical job-type mixes and the administrator's policies and can be recalculated in regular time intervals. One slice for each type is scheduled per time interval (since short jobs backfill into other slices in most cases, their slice is only scheduled if short jobs are waiting), and the slice times can be decided at the beginning of each interval. This permits controlling the resource allocation via different policies at different times of the day or via adaptive allocation which considers the current load of the machine [12]. In the context of this paper, the relative slice times are kept static.

Jobs per job type are scheduled in FCFS. Additionally, the typical backfilling is applied. Backfilling means that jobs can move ahead in the queue if they do not

delay other jobs as specified by the backfilling approach. Scojo-PECT can either use EASY or conservative backfilling. In the presented work, we use conservative backfilling which requires that none of the jobs in the queue are delayed.

Since the separation of jobs into different types is likely to increase the fragmentation because job sizes and job runtimes tend to be correlated, Scojo-PECT employs additionally safe non-type slice backfilling. This means that preempted or waiting jobs of a different type may be backfilled into a slice-with this backfilling only being valid until the end of the slice-if they do not delay any job of the slice type or of their own type according to the backfilling approach applied.

If setting time slices (resource shares) for equal service, Scojo-PECT provides similar service to medium and long jobs as standard space sharing with priorities but improves overall response times by about 50% by serving short jobs better [5].

The basic framework remains to be Scojo-PECT, and G-LOMARC-TS is applied per virtual machine. Only jobs of the same type are matched, though non-type slice backfilling is still applied. Thus, G-LOMARC-TS schedules jobs per virtual machine and does not even need to know about the existence of time slices. Groups are preempted and resumed as well as non-type slice backfilled like individual jobs. By representing this at job level, groups remain transparent to Scojo-PECT. FCFS is kept as basic scheduling order per virtual machine with constrained reordering as discussed above. The FCFS scheduling order and the compressed (keeping backfilled jobs in their backfill position even if jobs terminate earlier than estimated) conservative backfilling permit estimation of response times via simulation. As explained in Section 4.1, the estimated response times and the slack factor define the constraints.

4.8 Basic Job Creation

For the evaluation of our scheduler, we use the Lublin-Feitelson statistical workload model [8] which is the best available synthetic workload model (it includes power-of-two sizes, sequential jobs, correlations between runtimes and sizes, and varying inter-arrival times at different times of the day). Though the Lublin-Feitelson workload model was derived from statistical evaluation of 3 real-life workload traces, it generalizes the workload generation to the point that different machine sizes can be chosen. However, the Lublin-Feitelson model assumes that applications are run with 1 process per compute node though we need to model a hierarchical structure with multiple cores and subsequently the possibility of multiple processes per node. Thus, using the number of nodes as machine size would create a load which is too low. Multiplying the number of nodes by the number of cores per node would create a machine load which is too high because the additional cores add less performance gain than independent nodes [2]. Our goal is to create a workload with a similar load (utilization) and similar job/size characteristics as the original workload to have a similarly realistic model of the real world. In detail this means:

- Keep the runtimes of jobs the same, while letting jobs double or quadruple the number of processes by exploiting several cores per node or leaving the

process number unchanged, depending on the modeled self slowdown (see Section 4.9 for definitions of SL_{scr} and SL_{sno}). If the self slowdown of having 4 processes per node is less than a threshold MAX_SL (Table 6), the job size is quadrupled. If the self slowdown of having 2 processes per node is less than a smaller threshold $SELF\_SL\_2$ (Table 6), the job size is doubled. Thus, we adjust to both multiple cores per node and subsequent resource contention. Note that the modification of the workload corresponds to our model of space sharing as defined in Section 4.3 and used for the evaluation.
- Keep the percentage of serial jobs the same.
- Keep the percentage of power-of-two size jobs the same.

Note that the proper modification of the workload can be verified (which we did) by checking the average response times or average relative response times which should remain similar to the combination of the original machine and workload model if applying space sharing per virtual machine.

The rationale for our modification is that with more cores being available, users would likely run applications with more processes and tackle larger problem sizes. Moreover, since average runtimes were found to depend on the relative work submitted to the machine and not on the shape of the jobs, this is one of the feasible options of adjustment [12].

The detailed modification applied per job is described in the following formula ($nSize$ is the new size and $oSize$ is the original size):

$$nSize = \begin{cases} oSize & oSize = 1 || (SL_{sno} > SELF\_SL\_2 \\ & \&\& \; SL_{scr} * SL_{sno} > MAX\_SL) \\ oSize * 2/SL_{sno} & oSize > 1 \&\& SL_{sno} \leq SELF\_SL\_2 \\ & \&\& \; SL_{scr} * SL_{sno} > MAX\_SL \\ oSize * 4/SL_{scr}/SL_{sno} & oSize > 1 \&\& SL_{scr} * SL_{sno} \leq MAX\_SL \end{cases} \quad (7)$$

4.9 Slowdown Modeling

As mentioned above, competition on shared resources like memory, network, disk, caches (if cache shared among cores) causes slowdown. This not only applies if different applications share resources (*coscheduling slowdown* SL_{cos}) but also if processes of the same application share resources per node. Slowdowns can differ depending on whether the processes run on different CPUs (*node self slowdown* SL_{sno}) or different cores of the same CPU (*core self slowdown* SL_{scr}). For simplification, we include any relative runtime changes due to changing the number of nodes used by exploiting different numbers of cores/CPUs per node in the self slowdown. We also include any potential memory contention in the slowdowns (analysis of workload traces from [6] suggests that typically 95% of the jobs need $\leq 50\%$ of the memory per node, i.e. that memory contention is not a major issue if only 2 jobs are coscheduled). Slowdowns depend on the applications' characteristics in regards to the usage of resources and require proper slowdown metrics. Since resource usage characteristics are not available and would already require assumptions and since the slowdown metric goes

beyond the scope of this paper, we chose to directly model slowdowns statistically based on available experimental data.

To derive the statistical distribution of slowdowns, we took data from different experimental sources. Thus, we used data from [2] which investigates the performance gain from using dual-core vs. single-core AMD nodes in the Cray Red Storm system at Sandia National Laboratories. If comparing the normalized grind times of the PARTISN benchmark (the only benchmark with all data needed) for the same number of processes on single-core CPUs (T_{single}) and dual-core CPUs (T_{dual}), we obtain the slowdown as $SL_{scr} = 2 * T_{dual}/T_{single}$. The calculated slowdowns are shown in Table 2 (showing only machine sizes relevant to our simulation).

Table 2. SL_{scr} calculated from data for the PARTISN benchmark in [2].

Benchmark and configuration	32P/16N vs. 32P/32N	64P/32N vs. 64P/64N	128P/64N vs. 128P/128N	256P/128N vs. 256P/256N
Diffusion: 24^3 problems	1.31	1.46	1.30	1.42
Transport: 24^3 problems	1.05	1.19	1.00	1.16
Diffusion: 48^3 problems	1.61	1.51	1.48	1.57
Transport: 48^3 problems	1.29	1.15	1.09	1.21

Table 3. SL_{scr} and SL_{sno} for NAS benchmarks, as measured in [14]

Allocation of 16 processes to nodes	IS	EP	FT	CG	LU	BT*	MG	SP*
8N vs. 16N, multi-core per node	1.37	1.05	1.27	1.04	1.08	1.11	1.22	1.47
8N vs. 16N, multi-CPU per node	1.35	1.05	1.16	0.99	0.99	1.00	1.01	1.01

From [14], we also obtained data for SL_{scr} by investigating several NAS benchmarks. Though the experiments only involved 8 and 16 nodes, the data range is similar. The same paper also measured SL_{sno} shown in Table 3. Because the data for SL_{scr} is similar to Table 2, we consider SL_{sno} from Table 3 to be generally valid. Note that the data in Table 2 shows that slowdowns are not very sensitive to job size but more dependent on problem size and the problem itself. The latter two, however, are exactly what we capture with a statistical model.

For data in regards to SL_{cos}, we refer to [20] which investigates the coscheduling slowdown for combinations of NAS benchmarks and combinations of synthetic benchmarks (with different communication patterns, different communication percentages and different message sizes), run on 8, 32, and 64 nodes.

Taking the data from Table 2, Table 3, and from [20] (for SL_{cos}) as the typical spread of possible slowdowns, we calculated the distribution of slowdowns and

classified them into different ranges as shown in Table 4.[2] For example, in regards to SL_{scr}, 17% of the data above falls into the range [1.2, 1.3).

Table 4. Modeled distribution of slowdowns.

range	SL_{scr}	SL_{sno}	SL_{cos}
[0.9, 1.0)	0%	25%	0%
[1.0, 1.1)	25%	45%	68%
[1.1, 1.2)	17%	12%	17%
[1.2, 1.3)	17%	5%	7%
[1.3, 1.4)	13%	13%	3%
[1.4, 1.5)	17%	0%	2%
[1.5, 1.6)	8%	0%	1%
[1.6, 1.7)	3%	0%	1%
[1.7, 1.8)	0%	0%	1%

To simplify the modeling and the slowdown calculation, processes of the same job are currently allocated to cores of different CPUs per node, while processes of different jobs are allocated to the cores of the same CPU. This captures the most frequent cases that processes of the same job run better on different CPUs rather than on the cores of the same CPU (future extensions toward more differentiated performance considerations are possible). If there are N_{core} processes of the same job, they occupy all cores in a node.

5 Experimental Results

5.1 Experimental Set-up

We perform the evaluation via discrete event simulation with the workload model described in Section 4.8. Each test with the Lublin-Feitelson workload model is run with 3 random workloads (each 10,000 jobs) and results are averaged. The cluster used in the simulation has two dual-core CPUs (4 cores totally) per node. Table 5 shows the characteristics of the workloads. Workload $W1$ is the workload created with the original slightly adjusted Lublin parameters (since our scheduler currently involves 5% overhead,[3] we have reduced the workload in our scheduler vs. the original workload by 5% via slightly increasing the inter-arrival times). We also tested a busier Workload $W2$ which sets the α parameter in the inter-arrival time distribution to a smaller value and subsequently creates shorter inter-arrival times.

In regards to response-time estimation, we applied an adjustment by a factor of 0.75 to reflect that the estimates do not consider the benefits from non-type

[2] Minor adjustments of rounded values are done to obtain 100%.

[3] If jobs continue to run in the next slice, they do not actually need to be preempted but this reduction in overhead is currently not considered.

slice backfilling and coscheduling and jobs therefore run on average faster than estimated.

The parameters of Scojo-PECT are set to 30% relative time share for medium jobs and 70% relative time share for long jobs, 60 sec overhead per time slice for preemption/resumption of the jobs, and 1h intervals for scheduling one short (optional), one medium and one long time slice. Jobs are classified as short if their runtime is $\leq$ 10 minutes, as medium if their runtime is $\leq$ 3 hours, and as long otherwise.

Table 5. Workload characteristics.

Parameter	Value
$W1$ (normal load)	$\alpha = 10.33 \rightarrow Load = 10.6$
$W2$ (high load)	$\alpha = 9.83 \rightarrow Load = 13.0$
Machine size M	128
Percentage of short jobs N_S	64%
Percentage of medium jobs N_M	19.5% (54% of medium and long)
Percentage of long jobs N_L	16.5% (46% of medium and long)
Work of short jobs W_S	0.5%
Work of medium jobs W_M	26.0%
Work of long jobs W_L	73.5%
Percentage of serial jobs	24%
Percentage of jobs with power-of-two sizes	75%

To evaluate the performance of our algorithm, we compare to following approaches:

- SSP: Standard space sharing (only one job per node with 1, 2 or 4 processes, depending on the self slowdown as discussed in Section 4.3)
- $GLTS$: full group and time/space sharing G-LOMARC-TS

$$PPN = \begin{cases} 1 & jobsize = 1 || (SL_{sno} > SELF\_SL\_2 \\ & \&\& \; SL_{scr} * SL_{sno} > MAX\_SL) \\ 2 & jobsize > 1 \&\& SL_{sno} \leq SELF\_SL\_2 \\ & \&\& \; SL_{scr} * SL_{sno} > MAX\_SL \\ 4 & jobsize > 1 \&\& SL_{scr} * SL_{sno} \leq MAX\_SL \end{cases} \quad (8)$$

- CST: Only coscheduling and matchmaking two jobs, while taking the best suitable match
- $CSTWS$: Only coscheduling and matchmaking two jobs, while taking the first match

We also experiment with different variants of G-LOMARC-TS:

- FBO: Only matchmaking in the first block
- NS: No sorting per block

– NH: No sorting and no blocks, while selecting the first suitable group

All scheduling approaches use conservative backfilling to support prediction. Table 6 shows all scheduler parameters used in our experiments.

Table 6. Scheduler parameters used in the experiments.

Parameter	Value	Explanation
MAX_SL	1.25	Maximum slowdown that a job should experience
$SELF\_SL\_2$	1.12	Maximum self slowdown with 2 processes of a job per node
$MIN\_UTILGAIN$	0.45	Minimum utilization gain a group should achieve
$BLOCK\_SIZE$	16	Number of jobs per block
$MATCHED\_LARGER$	0.125	$X - Y \leq MATCHED\_EXCEED\_PRIM * Y$ if X is the sum of the node requirements of all matched jobs and Y is the node requirement of the primary job
$RUNNING\_RPIM\_NUM$	8	Number of running jobs which are considered as primary jobs
$MATCHED\_LONGER$	3,000	Maximum time in seconds by which a matched job can be longer than the the primary job in a group
F_{slack}	1.5	Maximum slack factor for a job
N_H	12	Threshold rating high-load phases

We used the following metrics for comparison:

– Average bounded relative response time[4](RR): response time in relation to pure runtime (without time slicing) while using cut-offs for very short jobs (only relevant for all-job evaluation)
– Core utilization U_{core} during high-load phases (see Section 4.6)

5.2 General Performance Results

The performance results (measured in RR) for G-LOMARC-TS compared to space sharing (SSP) and matchmaking for only two jobs (CST and $CSTWS$) are shown in Figure 6. The results show that the full algorithm of G-LOMARC-TS ($GLTS$) compared to SSP performs by 34.9% better for medium jobs, by 53.2% for long jobs, and 35.5% for all jobs. Note that this means that long jobs benefit more. Compared to matching only two jobs with best match (CST), the improvement is 19.4% for medium jobs, 16.1% for long jobs, and 13.9% for all jobs. Compared to matching only two jobs with the first suitable match

[4] The bounded relative response time is often called bounded slowdown. We avoid this term to avoid confusion with the slowdown due to resource contention.

($CSTWS$), the improvement is 28.6% for medium jobs, 31.5% for long jobs, and 23.2% for all jobs. This demonstrates that G-LOMARC-TS significantly improves average relative response times and that group matchmaking contributes significantly to the improvements.

Looking into the details of group matchmaking, we found an average of 2.5 jobs per group and about 1,500 (slightly depending on the concrete workload) groups being formed. Since the number of matched jobs is decided by the size of the primary job, on average, a job cannot match with many other jobs, and there are cases with only two jobs in a group as well as groups with more jobs. However, as discussed, we still obtain a significant improvement from group matchmaking. Since only medium and long jobs are coscheduled and they account for 36% of all jobs, this means that 41.7% of the jobs that are eligible for coscheduling actually run in a group.

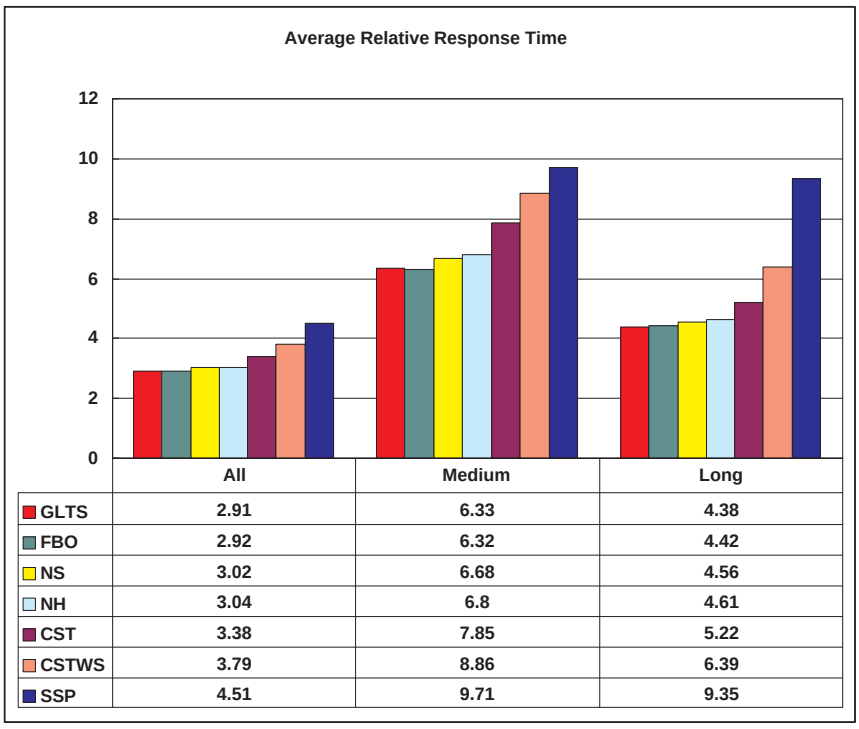
Figure 7 shows core utilization for high-load phases. We see that $GLTS$ improves utilization by 26.8% vs. SSP and by 6.0% and 10.1% vs. CST and $CSTWS$. This demonstrates that the source of the relative response time improvements is the increased core utilization in high-load phases and that the increase vs. SSP is significant.

In regards to fairness, the distribution of delays shows that on average jobs finish at their predicted response time. The 75% percentile is a delay factor of 1.3 for both M and L jobs and the 95% percentile is a factor of 1.4. Thus, the scheduler meets its goal of supporting fairness well.

5.3 Impact of Different Heuristics and Machine Load

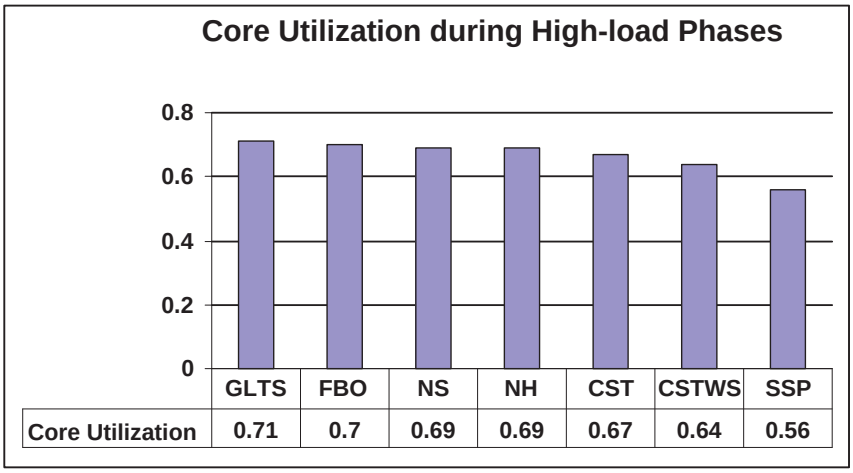
Figure 6 also includes results of several variants of G-LOMARC-TS. However, we do not see much impact from using the full matchmaking algorithm ($GLTS$) vs. simplifications which only match within the first block (FBO), do not sort (NS), or only selecting the first suitable group (NH). The reason is that the number of candidate jobs for matchmaking is less than 4 in 85% of the cases.

For Workload $W2$, the situation looks different. The results in Figure 8 show that $GLTS$ achieves the best results with optimum group size. NH does not depend on block size and is better than $GLTS$ with small and large block size for medium jobs but becomes significantly worse than $GLTS$ with optimum block size. FBO is much worse than $GLTS$ if the block size is small (because fewer jobs are considered) and becomes similar to $GLTS$ if block sizes are large enough. NS is especially worse than $GLTS$ for the optimum block size. Notable is that the optimum block size is 12 for both medium and long jobs. If the block size is too small, not enough matching candidates are available in the first block and more jobs are selected from the other blocks. If the block size is too large, jobs may be matched from the end of the first block. In both cases, jobs from further down the queue may move ahead and delay the jobs further up in the queue. Under Workload $W2$, the average group size is 2.6 (almost unchanged) but about 2,000 groups are formed which is about 25% more than for the basic workload. This is due to the fact that the waiting queues become longer and more matching candidates are available.



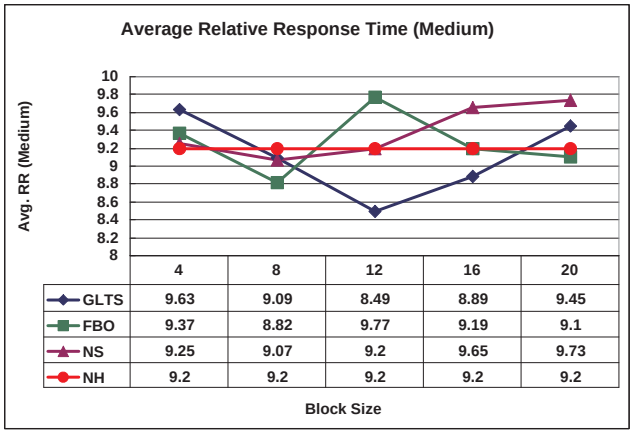
	All	Medium	Long
GLTS	2.91	6.33	4.38
FBO	2.92	6.32	4.42
NS	3.02	6.68	4.56
NH	3.04	6.8	4.61
CST	3.38	7.85	5.22
CSTWS	3.79	8.86	6.39
SSP	4.51	9.71	9.35

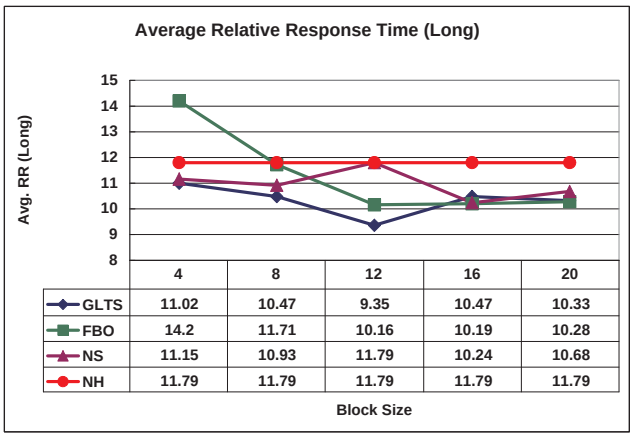
Fig. 6. *RR* for different schedulers and G-LOMARC-TS variants with Workload $W1$.



	GLTS	FBO	NS	NH	CST	CSTWS	SSP
Core Utilization	0.71	0.7	0.69	0.69	0.67	0.64	0.56

Fig. 7. Core utilizations during high-load phases for different schedulers and G-LOMARC-TS variants with Workload $W1$.





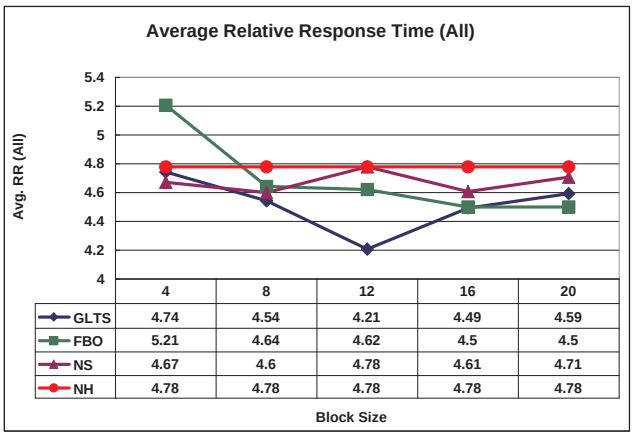
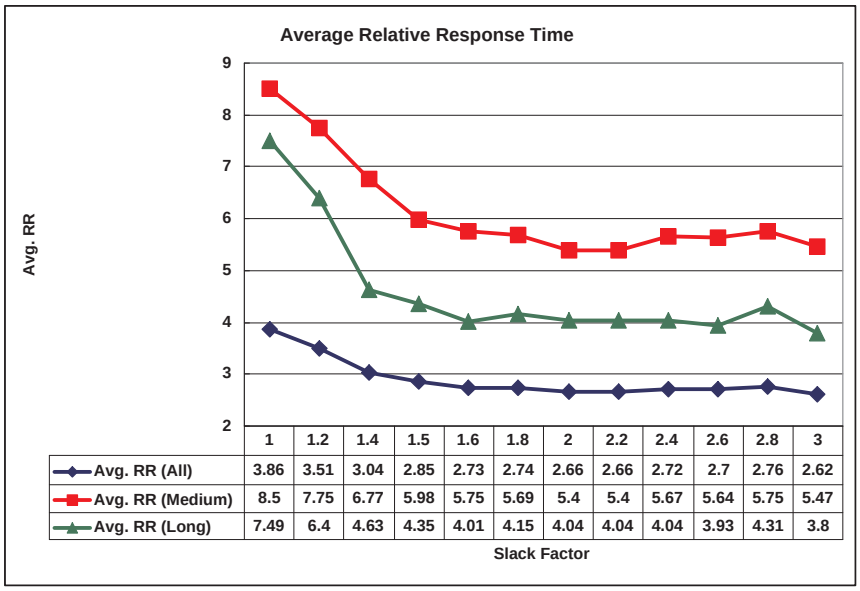


Fig. 8. Average relative response time for G-LOMARC-TS and different variants with different block sizes under Workload $W2$.

SPP cannot even handle Workload $W2$ and jobs queue-up as shown by an increased makespan-which is not the case for G-LOMARC-TS. Correspondingly, with SPP, the average relative response times become 17.5 (medium jobs), 36.74 (long jobs), and 10.65 (all jobs) which is much longer than G-LOMARC-TS. This demonstrates that our scheduler not only runs significantly better if the workload is normal ($W1$) but also can handle a much higher workload ($W2$) since the increased utilization makes it possible to run more jobs over the same time period.



	1	1.2	1.4	1.5	1.6	1.8	2	2.2	2.4	2.6	2.8	3
Avg. RR (All)	3.86	3.51	3.04	2.85	2.73	2.74	2.66	2.66	2.72	2.7	2.76	2.62
Avg. RR (Medium)	8.5	7.75	6.77	5.98	5.75	5.69	5.4	5.4	5.67	5.64	5.75	5.47
Avg. RR (Long)	7.49	6.4	4.63	4.35	4.01	4.15	4.04	4.04	4.04	3.93	4.31	3.8

Fig. 9. Average relative response time for Workload $W1$ with different slack factors F_{slack}.

5.4 Fairness vs. Utilization

Next we investigate the trade-off between optimization for highest utilization gain and fairness by running the G-LOMARC-TS with different slack factors. The results are shown in Figure 9. If the slack factor is low, more possible groups are rejected which leads to higher average relative response times. With increasing slack factor, more groups pass the fairness check and average relative response times improve. This is true up to a certain slack factor from which on the results become approximately equal. This can be explained by most suitable groups pass the check if the slack factor reaches a certain value. As can be seen

from the figure, the threshold is $F_{slack} = 1.5$ (used in all experiments above). Thus, larger slack factors than that certain value make no sense. Smaller slack factors may be chosen if fairness is rated higher than relative response times.

6 Summary and Conclusion

We have presented the G-LOMARC-TS scheduler which incorporates both space sharing and semi time sharing on clusters with multi-core nodes. G-LOMARC-TS employs group matchmaking and constraints the matchmaking by fairness to individual jobs. G-LOMARC-TS is integrated with the coarse-grain preemptive Scojo-PECT scheduler by applying G-LOMARC-TS per virtual machine managed in Scojo-PECT. Relative response times and core utilization are significantly improved with G-LOMARC-TS, and the scheduler can handle heavier workloads than space sharing per virtual machine, i.e. increases the saturation point. The results also demonstrate that group matchmaking contributes significantly to the improvements.

References

[1] Alves, F., Silva, B., Scherson, I.D.: Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler. Proc. 11th Symp. on Computer Architecture and High Performance Computing, Natal, Brazil (Sept. 1999)

[2] Brightwell, R., Underwood, K.D., Vaughan, C.: An Evaluation of the Impacts of Network Bandwidth and Dual-Core Processors on Scalability. Proc. Internat. Supercomputing Conference, Dresden, Germany (June 2007)

[3] Carrington, L., Wolter, N., Snavely, A., Bailey Lee, C.: Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications. Proc. DoD Users Group Conference, IEEE (2004)

[4] Chai, L., Gao, Q., Panda,D.K.: Understanding the Impact of Multi-Core Architecture in Cluster Computing–A Case Study with Intel Dual-Core System. Proc. CCGRID, IEEE (2007)

[5] Esbaugh, B., Sodan, A.C.: Coarse-Grain Time Slicing with Resource-Share Control in Parallel-Job Scheduling. Proc. High Performance Computing and Communication (HPCC), Houston, LNCS 4782, Springer Verlag (Sept. 2007)

[6] Feitelson, D.G., Rudolph, L., Schwiegelsohn, U., Sevcik, K.C., Parkson, W.: Theory and Practice in Parallel Job Scheduling. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS 1291, Springer Verlag (1997)

[7] Jung, C., Lim, D., Lee, J., Han, S.: Adaptive Execution Techniques for SMT Multiprocessor Architectures. Proc. ACM SIGPLAN PPoPP, Chicago, Illinois (June 2005)

[8] Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R.: An Empirical Study of Hyper-Threading in High Performance Computing Clusters. Proc. Linux HPC Revolution (2002)

[9] Lublin, U., Feitelson, D.G.: The Workload on Parallel Supercomputers-Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing, 63(11) (Nov. 2003) 1105-1122

[10] Moscibroda, T., Mutlu, O.: Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. Proc. 16th USENIX Security Symp., Boston (2007)
[11] Sabin, G., Sadayapan, P.: Unfairness Metrics for Space-Sharing Parallel Job Schedulers. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Cambridge, LNCS 3834, Springer Verlag (2005)
[12] Sodan, A.C.: Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability–First Experiences, Job Scheduling Strategies for Parallel Processing (JSSPP), Rome, Italy (May 2009)
[13] Sodan, A.C.: Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling–A Survey. Concurrency & Computation: Practice & Experience, 17(15) (Dec. 2005) 1725-1781
[14] Sodan, A.C., Gupta, G., Deshmeh, A., Zeng, X.: Benefits of Semi Time Sharing and Trading Space vs. Time in Computational Grids. Technical Report 08-020, University of Windsor, Computer Science (May 2008)
[15] Sodan, A.C., Gupta, G.: Time vs. Space Adaptation with ATOP-Grid. Proc. ACM Workshop on Adaptive and Reflective Middleware (ARM), Melbourne, Australia (Nov. 2006)
[16] Sodan, A.C., Lan, L.: LOMARC Lookahead Matchmaking for Multiresource Coscheduling on Hyperthreaded CPUs. IEEE Trans. on Parallel and Distributed Systems, 17(11) (Nov. 2006) 1360-1375
[17] Talby, D., Feitelson, D.G.: Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. Proc. Internat. Symp. on Parallel Processing & Symp. on Parallel and Distributed Processing (IPPS/SPDP), Puerto Rico, IEEE (1999)
[18] Weinberg, J., Snavely, A.: Symbiotic Space-Sharing on SDSC's DataStar System. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS 4376, Springer Verlag (2007)
[19] Wiseman, Y., Feitelson, D.G.: Paired Gang Scheduling. IEEE Trans. Parallel and Distributed Systems (2003)
[20] Zeng, X., Shi, J., Cao, X., Sodan, A.C.: Grid Scheduling with ATOP-Grid under Time Sharing. Proc. CoreGrid Workshop on Grid Middleware, Dresden, Springer Verlag (June 2007)