2009

G-LOMARC-TS: Lookahead group matchmaking for time/space sharing on multi-core parallel machines

Xijie Zeng
University of Windsor

G-LOMARC-TS: Lookahead Group Matchmaking for Time/Space Sharing on Multi-core Parallel Machines

By

Xijie Zeng

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2009

Your file Votre référence
ISBN: 978-0-494-57554-3
Our file Notre référence
ISBN: 978-0-494-57554-3

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Parallel machines with multi-core nodes are becoming increasingly popular. The performances of applications running on these machines are improved gradually due to the resource competition in each node. Researches have found that coscheduling different applications with complementary resource characteristics on the same set of nodes (semi time sharing) may improve the performance. We propose a scheduling algorithm G-LOMARC-TS which incorporates both space and semi time sharing scheduling methods and matches groups of jobs if possible for coscheduling. Since matchmaking may select jobs further down the waiting queue and the jobs in front of the queue may be delayed subsequently, fairness for each individual job will be watched and the delay will be kept within a limited bound. Several heuristics are used to solve the NP-complete problem of forming groups. Our experiment results show both utilization gain and average relative response time improvements of G-LOMARC-TS over other several scheduling policies.

To my husband, Yu Chen

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor Dr. Angela C. Sodan. Without her extensive guidance and constant encouragement, I could not finish this thesis. I would also like to thank all committee members, Dr. James Gauld, Dr. Asish Mukhopadhyay and Dr. Peter Tsin for their valuable time and comments.

My special thanks go to my husband, Yu Chen, for his support and good advices during the two and half years' graduate study.

I would like to thank Wei Jin and Xiaorong Cao for the happy experience during the time I worked with them.

Last, but not least, I would like to thank all of my friends for all the help and support during the completion of this thesis.

Table of Contents

List of Tables

List of Figures

1. Introduction

Job scheduling in parallel systems is a hotly studied subject in High Performance Computing (HPC) field. A job, in this context, is an application with one (serial job) to multiple (parallel job) processes. Job scheduling refers to *when* and *where* a job is scheduled to run on a parallel system.

Three metrics are usually used to evaluate a scheduling strategy. From the perspective of users, *response time* represents how fast a user's request is served. It is the time period between the job submission and termination of execution. Usually, only average response time is reported instead of each individual value. It may happen that some individual jobs have poor response times and are severely delayed while the average response time is very good. Thus, *Fairness* is usually applied to each individual job to check whether they are fairly or unfairly treated. Fairness check is to set up a threshold which is the maximum slack factor when comparing a job's response time and its estimated response time produced upon submission. From the system's view, *utilization* denotes how efficiently the system resources are been used. It is the ratio of used resources and total resources during a time period. High utilization (keeping the system busy) may lead to shorter response times but not definitely. There is a trade-off between these two factors. Scheduling strategies may take some metrics as their main goals and the other as constraints.

Two basic types of job scheduling approaches are time sharing and space sharing. With time sharing, multiple jobs are allocated to run on the same set of processors. Multiple processes share a processor with time slices. The advantage of time sharing is that jobs can start to run sooner (maybe with a shorter response time eventually). However, time sharing suffers from context switch overhead when a processor suspends the current running process to schedule another one to run. Synchronization among the processes of the same job is another problem of time sharing. Processes of a job communicate with each other by sending/receiving messages. However, scheduling which process to run is independent among processors. This feature makes one process waiting if it wants to communicate with another process in a different processor which is not scheduled at the same time. With space sharing, processors

are divided into groups with each group exclusively allocated to a job which means a job can continue running until finish without being suspended. Space sharing is easy to implement with no overhead of context switch. Unfortunately, fragmentation is the main problem of space sharing due to its inefficient packing scheme. Some scheduling methods are proposed to improve the performance of time and space sharing, e.g. implicit/dynamic coscheduling to time sharing; backfilling, a job is scheduled to run out of its original FCFS (First Come First Serve) order to fill the "hole", to space sharing; the combination of both time and space sharing. Section 2 will discuss these methods in detail.

Multi-core processors are the most important trend in the contemporary computer architectures. As a result, an increasing number of parallel systems, e.g. clusters, facilitate their nodes with multiple multi-core processors to make their system more powerful, like Sharcnet. Unfortunately, the performances of applications running on these systems are not improved in the same scale as the increase of the cores due to the competition on shared resources such as the network, memory and potentially cache in each node. Studies in [28][29][38] have shown that allocating two applications with different resource usage characteristics on the same set of nodes may achieve better results. We call such resource sharing scheme as semi time sharing. With semi time sharing [28] (the concept was first proposed in [29]), processes from different applications are allocated to the different cores of a node. Processes per node exclusively occupy cores by one core per process without sharing with time slices and share other resources such as the memory, disk, network and maybe the cache. Selecting jobs with complementary resource requirements is critical to semi time sharing.

This thesis proposes a job scheduling algorithm named G-LOMARC-TS for a cluster architecture with multiple multi-core processors per node. G-LOMARC-TS tends to improve system utilization and jobs' average relative response time (the ratio of response time and runtime of a job) by combining both space sharing and semi time sharing scheduling approaches. G-LOMARC-TS employs group matchmaking instead of paring only two jobs and permits partial reordering of waiting jobs in order to find suitable matches.

G-LOMARC-TS applies a fairness check to individual jobs to make sure that no jobs are severely delayed due to reordering and slowdown caused by coscheduling. Fairness is applied by allowing a maximum slack factor when comparing a job's response time and the original estimated response time obtained upon job submission. Heuristics are used to simplify the NP-complete problem of forming groups among multiple waiting and/or running jobs. Utilization gain is used as a metric to select optimal groups. At the same time, special processing is applied for serial jobs since there may be bursts of serial jobs with similar characteristics in close time proximity.

The rest of the thesis is organized as follows. Background issues are discussed in Section 2. Section 3 introduces multi-core processors and multi-processor nodes. G-LOMARC-TS job scheduling algorithm is described in detail in Section 4. Section 5 presents the simulation, experiments and results analysis. Finally, the conclusion of the thesis is made in Section 6 and we discuss a little about future work in Section 7.

2. Background Issues

Job scheduling in distributed high performance platform has been studied for a long time. From the perspective of processor allocation, the scheduling approaches are divided into 2 classes: space sharing and time sharing. With space sharing, processors are partitioned into groups and each group is exclusively assigned to a job. With time sharing, each processor is shared by multiple processes from different jobs. (These processes may run in different time slices in a round robin fashion.)

Gang-scheduling (GS, also known as explicit coscheduling) is a well known time sharing method which is first proposed by Ousterhout in [21]. In GS, all processors are time-sliced in a coordinated manner and each time slice acts as a virtual machine. All processes of a job are globally synchronized in a time slice which facilitates the communication among processes. There may be multiple jobs to share the space in each slice. The evaluation in [9][14] showed gang-scheduling to be a very promising approach which greatly reduced response time and slowdown. Similarly, the study in [8] indicates that gang-scheduling outperforms non-preemptive policies such as FCFS and FCFS-backfill. However, some recent researches [40] point out that there will be fragmentations since freed processors in one time slice can not be allocated to the jobs in another time slice because of the independence of time slices. Thus, the study in [40] proposes a technology named job re-packing which tries to migrate the job execution among different time slices so that small fragments of idle processors from different time slices can be combined together to form a larger and more useful one in a single time slice. Gang-scheduling does not guarantee the completion time of real-time jobs which have requirements to be finished before a specific time. In [39], jobs are divided into 2 types: real-time jobs and best-effort jobs which require to be completed as soon as possible. The Ousterhout matrix [21] rows are split into 2 sets, corresponding to the 2 types of the jobs, with the relative proportion of the 2 sets of rows conforming to a fairness ratio which helps to guarantee the completion of real-time jobs.

Though gang-scheduling is popular it has its own imperative drawbacks. All the processes of a job are synchronized, so it is impossible to avoid communication and I/O latency. At the

same time, the performance of gang-scheduling will be limited due to the memory constraints if all the jobs reside in the main memory. In order to overcome these limitations of gang-scheduling, several non-explicit coscheduling approaches which approximately coordinate communicating processes of a job, are proposed in [1][2][10][11][12][13][22][36][37]. With spin-blocking [2], after the sender sends out a message, it will busy wait for a period of time hoping that the receiver will be scheduled soon. If it does not receive the response from the receiver within the time period it will be blocked. Spin-blocking is used in cooperation with the local Unix scheduling system which is round-robin and priority-based. Similarly, the plain Linux scheduling policy is improved in [11] by adjusting the length of time slice based on whether there is local jobs and on the cache miss rate.

Some kinds of communication driven coscheduling technologies are fully studied in [1] [10][12][13][22][36][37]. Except for the local message arrival information, some other mechanisms for sharing the load, status and resource allocation information among nodes are also provided in [12][13][37]. Cooperative Coscheduling (CCS) proposed in [12][13] employ event based notification mechanism to share the memory, processor, I/O and workload information among nodes. In [37], however, these global synchronization and load imbalance information are inserted in an out-going message to inform the remote communication partner of its status. The results obtained in [13] reveal that CCS efficiently exploits idle resources and consequently obtains a better speedup than implicit and dynamic coscheduling. Unlike other communication-driven coscheduling algorithms implementing synchronization only on receiver side, the approach in [1] achieves coordination running by optimizing at both sender and receiver sides. Except for the receiver side gaining higher priority to be scheduled on message arrival, on the sender side, the sender spins for a pre-defined amount of time to wait for a send complete event. If a send is completed within that time, the sender remains scheduled, hoping that the receiver will quickly be scheduled and a response can soon be received. However, if a send is not completed within this spin time, the sender process will be blocked. As soon as the send completes, the sender process will be woken up and ready to be scheduled. In prior coscheduling methods, the schedulers are responsible for

boosting a process's priority and schedule it if there are unprocessed messages. However, in [36], processes will detect coming messages by themselves. The local scheduler periodically schedules all the jobs for a brief period of time to detect any message arrival. If a process detects a coming message, it sends a schedule request to the scheduler. The scheduler guarantees a communicating process to run a given time period without being preempted by another job. In order to achieve flexible coscheduling, jobs are divided into 3 classes in [10] based on their communication behaviors (CS: the processes communicate often and the machines run effectively; F: the processes have enough synchronization requirements but the machines are not effectively used because of load imbalance; DC: the processes rarely communicate.) CS processes are always coscheduled and should not be preempted. F processes need coscheduling but are preempted if synchronization is ineffective. All coscheduling technologies described above make decisions in a per-message basis, while the method in [22] accumulates and buffers the messages and only do real communication every pre-defined period of time in order to minimize the communication and scheduling overhead.

Non-explicit coscheduling algorithms solve the problem brought by gang-scheduling, however they have the overhead of context switch which occurs when a job is preempted and another job is scheduled to run. The researches in [19][26][27][29][33][35] try to coschedule multiple processes/threads on different logical processors of a hyperthreaded processor or of a multi-core processor or on different processors of a node. Then multiple processes from the same job or different jobs can run concurrently without context switch. The study in [35] investigates the effectiveness and slowdown of coscheduling two kinds of processes together in one node. These two kinds of processes are sensitive to two different resources (main memory, cache, processor or I/O). The result shows that there is only a minor or even no slowdown for most cases but with an exception. When main-memory and cache sensitive processes are coscheduled together, there is a significant degradation. A symbiotic space sharing scheduler is proposed in [35], which divides the node with 8 processors into 2 halves, the top and the bottom. The top half is designated to execute memory-sensitive (including 3 levels of caches) jobs and the bottom half for the other jobs. The scheduler obtains a speedup of 1.2 for makespan and a doubling of system utilization. Consequently, it indicates that

processes bound to different resources can be coscheduled in a node without interfering with each other that much. Similarly, the study in [26][27][29][33] investigates the effect of coscheduling two processes/threads on logical processors of a hyperthreaded processor. In [27][29], when the first job in the waiting queue is scheduled to run, the scheduler searches for a matched job in the waiting queue or among currently running jobs for coscheduling. The two coscheduled jobs must be both sensitive to processor or to different resources. The research in [19] does experiments on dual-core CMP (Chip Multi-Processors) of dual-threaded SMT (Simultaneous Multi-Threaded) processors. It not only addresses the contention on some specific resources, but also points out the sensitivity of a thread to the contention for some resources which is based on the assumption that knowledge regarding the load on the resource is less important than knowing the degree to which each thread's performance is impacted by a reduction in resource availability.

Some hybrid algorithms which combine both space and time sharing technologies are introduced in [6][15][16][17][32][34]. In [15][16], the approaches select some nodes for jobs using the uniform node selection policy which combines communication and computation sensitive jobs in the same nodes and they run jobs in an ordered manner, whenever possible. Ordering the jobs means to launch processes making up a pair of parallel jobs in the same set of nodes, balancing the workload across the cluster. With time sharing, it exploits the Cooperative Coscheduling [11][12] to coschedule allocated jobs in each node. Both [6] and [17] employ tree architecture to implement combination of space and time sharing. In a tree, each node represents some number of processors. If the root of the tree has two child nodes and if the root represents the whole processors of the system, each of the child nodes represents half of the processors in the system, and so on. Consequently, among nodes it is space sharing and within each node it is time–sharing. In [17], each node is attached with a job running queue. However, in [6], all jobs are in the running queue of the root node originally. The architecture can obtain self-balance by assigning jobs from the root node to the bottom nodes along the branches. Jobs are divided into two priority classes in [32]. High-priority jobs (primary jobs) are scheduled in a traditional space sharing manner, consuming resources of the processors where they are allocated to for execution.

Low-priority jobs (guest jobs) are scheduled on the same processing nodes as the high-priority jobs (time sharing with primary jobs) to consume the underutilized resources. In [34], jobs share the machine in a FCFS order with space sharing. However, the number of processors allocated to a job may be less than the process number of the job. Within each job, the processes time share the allocated processors. Here, the time sharing happens within a job instead of among jobs.

3. Multi-core Processor and Multi-processor Node

We are entering the multi-core era as single-core processor technology meets its complexity and speed limitation. To speedup a single-core processor, the traditional method is to increase frequency. However, two of the bad consequences are over-heat and power consumption which make it low cost-to-performance efficiency to raise clock rate. The move towards multi-core processor is the most important trend in the microprocessor architecture. In the context of this thesis, a core represents an execution unit which contains all the components required for instruction processing; while a processor is a physical chip or a die.

Major microprocessor vendors are combining multiple cores into a processor. Dual-core, tri-core, quad-core processors are becoming increasingly popular in personal computers; while processors with even near one hundred cores are spread out in industry usage. Examples are AMD Athlon 64 serial (2 cores), Opteron (2/4), Phenom (3/4); IBM POWER4/5 (2), Xenon (3) used in Microsoft Xbox 360, PowerPC 970 (2) used in Apple Power Mac G5; Intel Core Duo (2), Itanium 2 (2); Sun Micorsystems UltraSPARC IV (2), UltraSPAC T1 (8). Figure 1 shows architectures of a single-core, dual-core and N-core processors respectively.

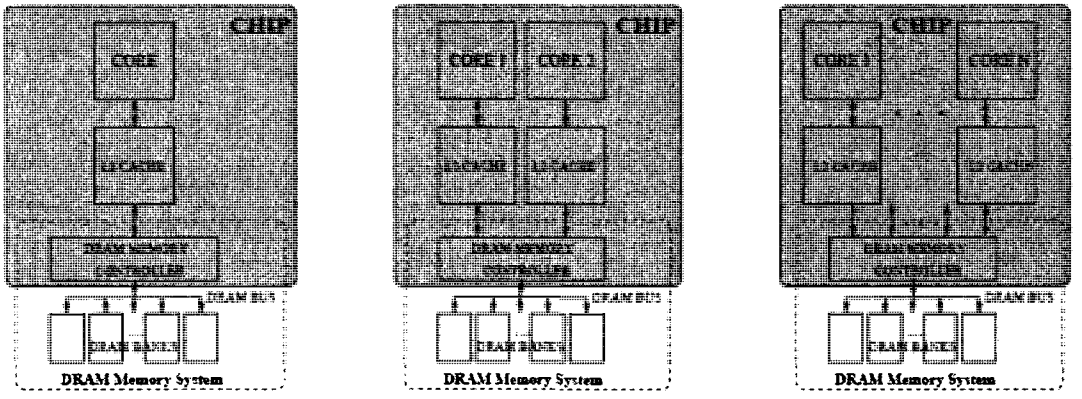


Figure 1. Architecture of a single-core system (left), a dual-core system (middle), an N-core system (right) (from [20]).

Due to the development of multi-core processors, clusters equipped with a number of these processors are a growing industry trend. In this thesis, we assume that our target machine is a cluster with multiple multi-core processors per node. Though multiple processors and

multiple cores per node can significantly increase performance, they do not simply multiply the performance by the number of cores due to the contention effects on shared resources. Processes running on the same node share the network, disk, and the memory. Processes running on the same processor additionally share the memory access paths and potentially the cache. In regards to the cache, some multi-core processors share the L2 cache (such as the Intel Core Duo and IBM POWER4/5), whereas other multi-core processors have private L2 caches per core (AMD Opteron, Intel Itanium, IBM POWER 6 and Ultra SPARC). If an application is improperly programmed or "non-matching" applications are coscheduled in a processor, the performance of the application will be greatly degraded compared to the single-core case. The experiments in [20] demonstrate that the current multi-core memory system is vulnerable to a new kind of attack named Denial of Service because the memory is "unfairly" shared among the applications running on multiple cores. A thread with particular memory access pattern may prevent the memory request of another thread to be served. The cause of the problem is that memory requests are served in FCFS order regardless of the fact that from which thread the request is submitted. The data shows that an application may slowdown another application 2.9 times while no much impact on itself. We model these contention effects as application slowdowns (Section 4.10), and differentiate between processor and core slowdowns, with processor slowdown typically being lower.

In the few cases where resource sharing among processes of the same application provides a benefit, the slowdown would turn into a speedup. Communication among the processes of the same job running on the same node is accomplished by the sharing of memory and cache. The cost is much less than that of inter-node communication. [5] observes that 50% messages are transferred through intra-node communication. It also shows that applications and middleware which are multi-core aware improves execution time by up to 70%.

4. G-LOMARC-TS

G-LOMARC-TS is proposed as an extension of LOMARC [27][29]. LOMARC only coschedules two jobs and supports only semi time sharing (no space sharing). With LOMARC, each job can only use one hyperthreaded CPU per node (the other one may be idle or used by another job). LOMARC uses a simple metric which assumes the same slowdown for the two coscheduled jobs and considers how much faster the two jobs run by coscheduling compared to individually scheduling. LOMARC is implemented as a priority based scheduler. G-LOMARC-TS coschedules groups of jobs instead of only two jobs and employs both space sharing and semi time sharing. Each job can exploit multiple cores per node. An advanced metric which considers the resource usage and interactions among multiple jobs is used to select optimal groups. G-LOMARC-TS also applies fairness check to make sure that no individual jobs are delayed severely. G-LOMARC-TS is implemented based on a coarse-grain preemption scheduler Scojo-PECT [7].

4.1. Assumptions and Goals

G-LOMARC-TS job scheduling algorithm is based on the following assumptions.

- In each node, there are multiple multi-core processors which make it possible to coschedule multiple processes per node. In our simulation and experiments, we provide results for clusters with 2 dual-core processors (totally 4 cores) per node.
- Jobs are submitted with processes instead of threads which make it flexible to schedule the processes to different nodes/processors/cores, e.g. MPI applications.
- The workload only includes rigid jobs with fixed number of processes (job size). No moldable jobs (size can be changed when the job starts to run) or malleable jobs (size can be changed at startup time and during the execution) will be considered.
- When a job is submitted, the user provides the process number (job size), runtime estimation and sufficient characteristics information to calculate slowdowns. As to the size, it is obvious that the user should have the knowledge about how many processes the job has. As to the runtime and characteristics, there are two methods to get the information. The one is that the user can collect these data by previous executions of the job. The

second one is that the user can run the job with a sample input and analyze the sample execution to get the estimation of runtime and resource requirements.

- There are a large number of serial jobs and many parallel jobs with power-of-two size in the workload. This is observed in many practical parallel system logs [18]. Serial job is the one with only one process while parallel job has at least two processes.
- There may be bursts of jobs with similar characteristics in close time proximity, e.g. a user may submit a job multiple times with different but similar parameters.

G-LOMARC-TS is designed with following goals in mind:

- Provide a flexible job scheduling algorithm which takes advantage of both space and semi time sharing. A job can exclusively occupy a set of nodes and all cores per node with space sharing. While, with semi time sharing, multiple jobs with complementary resource requirements share a set of nodes with different jobs executing on different cores per node. The current machine load, a job's runtime and utilization gain (Section 4.7) decide whether the space or semi time sharing is used.
- Lookahead in the waiting queue to find suitable matches which, as a result, will partially reorder the queue (originally it is in the FCFS order). So fairness constraint will be imposed to make sure that no individual jobs are overly delayed.
- 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.
- Improve system utilization, typically the core utilization (more cores per node are used, fewer idle), during high-load phases to reduce average relative response times and provide better services to users. The number of cores is increased per node but not with other resources such as the network, disk and the memory. If only one process is scheduled per node, only one core is used and the other idle cores are wasted. If we blindly schedule multiple processes to run on all cores per node without considering the contention on other resources among the processes, the performance may be greatly degraded due to the shared resource competition. G-LOMARC-TS tries to make good use of resources by matching jobs with complementary resource requirements to reduce the contention.

4.2. Scheduling Objectives

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 [30]. Note, that overall utilization remains the same under different scheduling policies if the workload remains the same, as long as the scheduler can handle the workload (i.e. jobs won't queue-up). Schedulers with less utilization support will only delay jobs during high-load phases and the delayed jobs will be scheduled when next low-load phase comes (and this is the reason why the jobs do not queue-up in the long run). This also means that schedulers with high utilization support may not delay jobs during high-load phases and make the machine idle or very lowly loaded during low-load phases.

Thus, we use utilization improvement as the primary objective and fairness as a constraint. This requires us to formally define high-load phases, core and node utilization, utilization gain, fairness, and the scheduler impact on fairness. Terms used through the thesis are listed and explained in Table 1.

S_i	Size (whole process number) of Job i
T_i	Runtime of Job i when scheduled individually with one process per node
PPN_i	Number of processes for Job I per node
$T_{makespan}$	Total runtime of the whole workload
M	Number of nodes in the cluster (machine size)
N_{core}	Number of cores per node
U_{core}	Core utilization
U_{node}	Node utilization
U_{gain}	Utilization gain resulted from the comparison of node usage
$R_{est,i}$	Estimated response time of Job I calculated at submission time in FCFS order
R_i	Response time of Job i
F_{slack}	Slack factor used for fairness check
$Phase_H$	High-load phases

Table 1. Terms used throughout the formulas in the thesis.

We define N_{wait} as 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 system status changes (due to job submission, job termination and slice switches, see Section 4.8), and t_l, t_j and t_k are certain time points of status 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:

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

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

$$U_{node} = \Sigma_{i \text{ in all time periods}} (tp_i * un_i) / (T_{makespan} * M) \quad (2)$$

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

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

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} (Section 4.7) is used to measure the efficiency of different groups (Section 4.5) and calculated in saved usage of nodes. Nodes are saved via increased core utilization. Note that, however, core utilization itself does not make a suitable metric because it does not consider the fact that the jobs' runtimes are extended due to the scheduling/coscheduling of multiple processes (from the same job or different jobs) per node. Therefore, high core utilization does not necessarily mean a good matchmaking/group. By pursuing high utilization gain, we will achieve both high and effective core utilization.

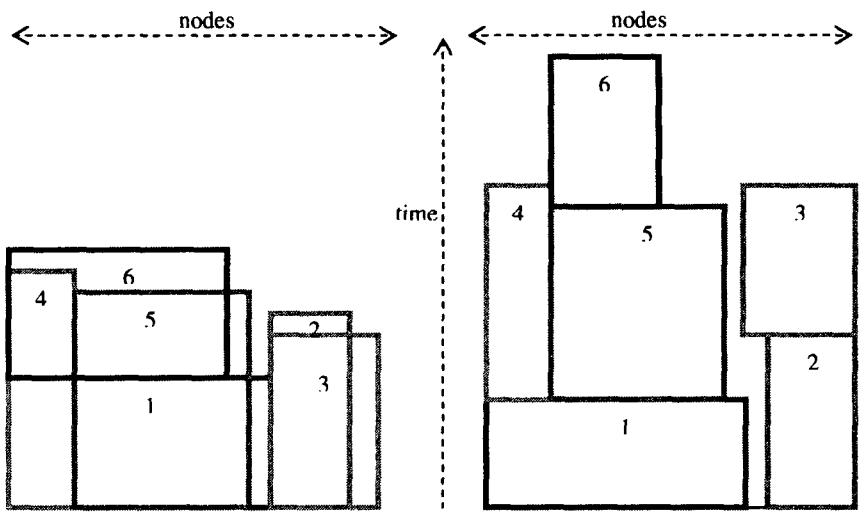


Figure 2. Utilization gain via increased core utilization for Job 1 to Job 6 with G-LOMARC-TS (left) and space sharing (right).

There exist many different considerations for fairness in the literature [3][23][24][25]. 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 calculating the response-time changes (R_i compared to $R_{est,i}$) which happen due to our effort for utilization improvement and by permitting limited delays (slack F_{slack}) vs. $R_{est,i}$:

$$R_{max,i} = F_{slack} * R_{est,i} \quad (4)$$

Finally, we define our objective by maximizing core utilization during high-load phases with $R_i \leq R_{max,i}$ which is approximated by individual decisions, heuristics and utilization gain per decision.

Figure 2 shows how high and effective 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, PPN is 2 with G-LOMARC-TS and 4 with space sharing; i.e, the node requirements are different under the two scheduling schemes).

4.3. General G-LOMARC-TS Scheduling Ideas

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 processors/cores per SMP (Symmetric Multi-Processor) node of a cluster are allocated to different jobs as illustrated in Figure 3. Processes per node run concurrently on different processors/cores with no overhead of context switch while standard time sharing suffers. 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.

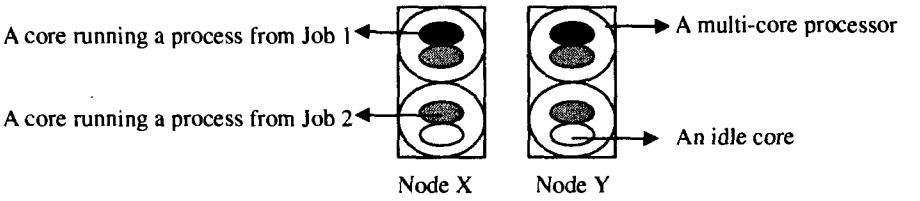


Figure 3. Coscheduling processes of different parallel jobs per node.

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 with semi time sharing, i.e. coscheduling processes from different applications on the same nodes, see Figure 3. Semi time sharing is aimed to highly utilize the multiple execution contexts (processors/cores) in each node and make such coscheduling effective. Consequently, job combinations with high complementary resource usage should be created. This does not typically 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 some running and waiting jobs. This is the reason why we need to include the fairness criterion as described in Section 4.2 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 [27]. 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.

4.4. Time vs. Space Scheduling

In the following, we explain the space sharing and semi time sharing options of G-LOMARC-TS in more detail. Both 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. 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 (Section 4.10) caused by resource contention of processes of the same job. If the self slowdown is severe, fewer processes per node per job are meaningful. If the self slowdown is low or there is even speedup, more processes per node per job are better.

In our scheduler, short jobs are only scheduled via space sharing because short jobs are not considered worth the effort of matchmaking due to their short runtimes. 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 it obtains the best runtimes per job. Otherwise, decisions between space and semi time sharing are made adaptively when trying to schedule the first job in the waiting queue. In fact, the decision depends on which option provides better utilization gain. Two special cases are that 1) 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 using semi time sharing 2) and if no good matches are found space sharing is the only choice.

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 (Section 4.10) caused by processes of different jobs, it may gain more with semi time sharing.

4.5. Scheduling Algorithm

The core of the G-LOMARC-TS scheduling algorithm is shown in Figure 6. 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 with other waiting jobs or with running jobs. In detail, the first waiting job can be scheduled/coscheduled in one of the following ways (one *primary* job and one or multiple *matched* jobs constitute a group, see Section 4.6):

- Case 1 (semi time sharing) in Figure 4: 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).

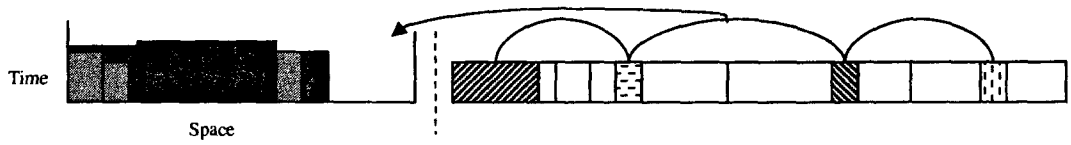


Figure 4. Case 1 (semi time sharing).

- Case 2 (semi time sharing) in Figure 5: coscheduled in a group with the first waiting job as the primary job and several running jobs as matched jobs (one waiting : multiple running).

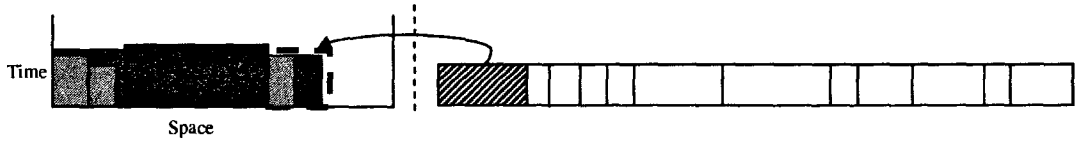


Figure 5. Case 2 (semi time sharing).

- Case 3 (semi time sharing) in Figure 7: 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.

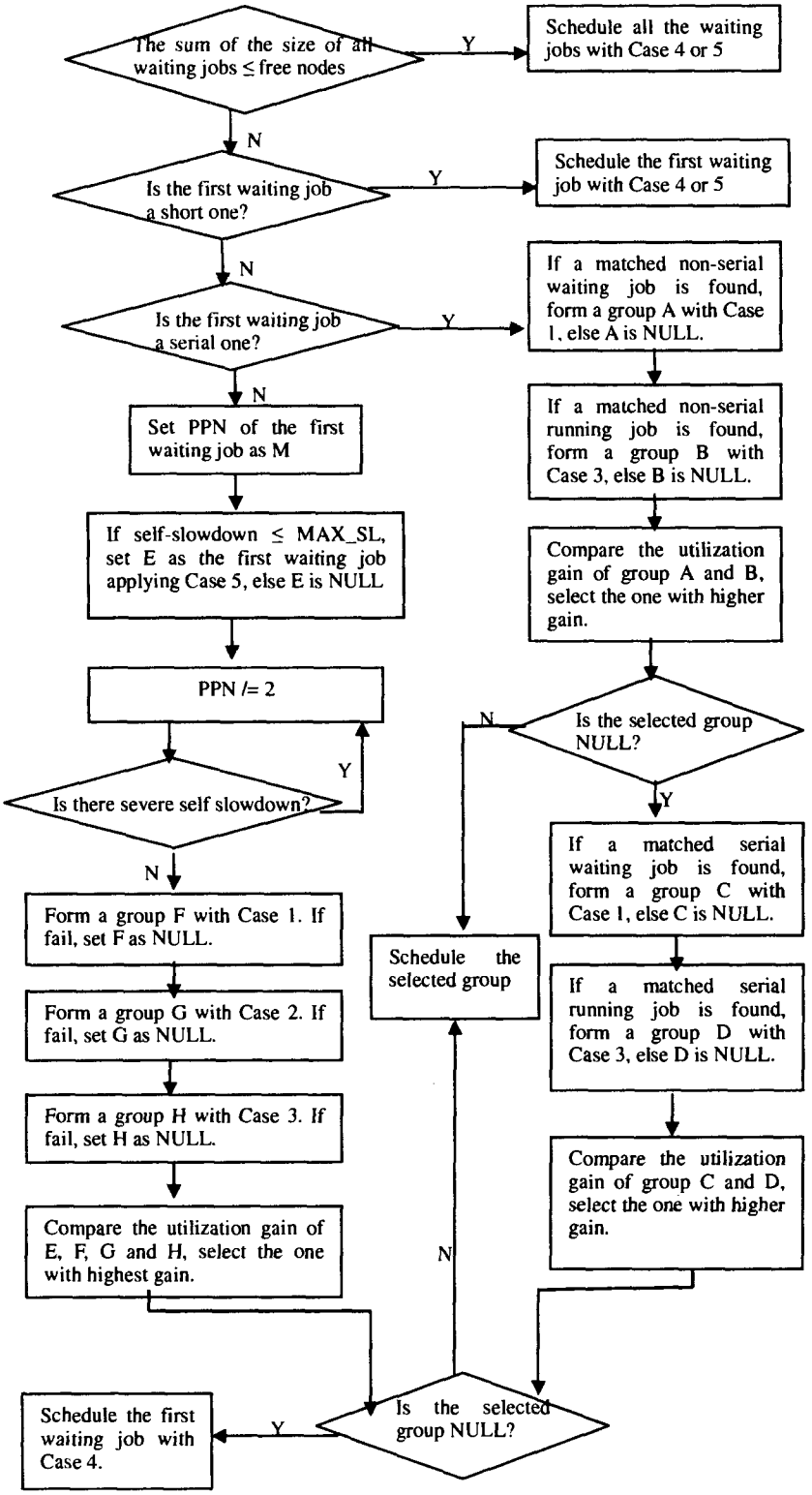


Figure 6. Flow chart for core scheduling algorithm.

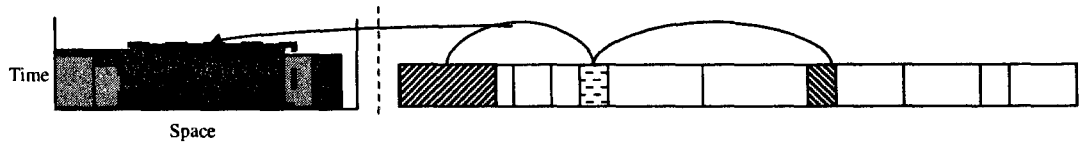


Figure 7. Case 3 (semi time sharing).

Specifically, G-LOMARC-TS scheduling algorithm works with following steps:

1. If the workload is low which means there are enough free nodes for all current waiting jobs to be scheduled one process per node individually with space sharing, all jobs in waiting queue will be scheduled with Case 4 or 5 depending on the self slowdowns of the jobs. The lower the self slowdown, the more processes per node.
2. If the first waiting job is a short one (Section 4.8), the job will be scheduled with space sharing (Case 4 or 5) since a short job finishes soon and does not deserve a coscheduling. In our simulated workload, there are more than 60% short jobs.
3. If the first waiting job is a serial one (the job with only one process), we handle it specially. According to the Lublin-Feitelson workload model [18], 24% of the jobs are serial. We assume that if there is one serial job in the waiting queue, there may be burst of serial jobs following. To process it, we form two groups. In one, we search for a parallel waiting job as the primary job and some serial waiting jobs including the first one as the matched jobs (Case 1). In the other group, we select a parallel running job as the primary job and some serial waiting jobs including the first one as the matched jobs (Case 3). Then, the group with higher utilization gain is selected to be scheduled.
4. We try to form 3 coscheduling groups with Case 1, Case 2 and Case 3 respectively. We compare the utilization gains (Section 4.7) of the 3 groups and space sharing of Case 5 and select the one with highest gain to schedule.
5. If the first waiting job can not be scheduled with previous steps, the scheduler tries to schedule it with Case 4.

A similar algorithm is applied when attempting to backfill jobs. However, only Case 1, Case 4, and Case 5 are applied. Note that a job may be matched more than once over its runtime if its group is disbanded and the job becomes 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 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.

4.6. Group Formation

A *group* contains several jobs (at least 2) which have complementary resource requirements. G-LOMARC-TS coschedules jobs in a group to the same set of nodes to effectively use the system resources. In fact, a group is composed of a *primary* job and one or multiple *matched* jobs. Thus, the relationship among the jobs in a group is one (primary job) to multiple (matched jobs) (1 : n). Figure 8 shows an example of a group with 4 jobs (the "background" big one as the primary job, the 3 "front" small ones as the matched jobs). Per node, only two jobs are coscheduled with semi time sharing (one is the primary job, the other one is one of the matched jobs.).

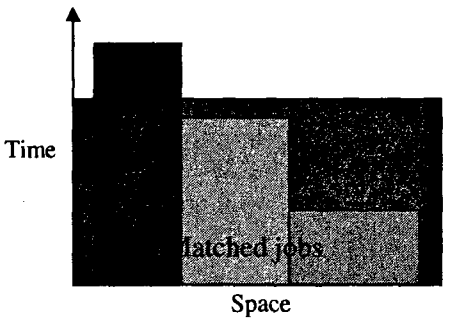


Figure 8. A group with 4 jobs (1 primary job, 3 matched jobs).

There are various possibilities to combine jobs with different runtimes and sizes into groups. Additionally, slowdowns depend on the job combination. Thus, we are facing an NP-complete problem. To make the problem tractable, we apply intelligent heuristics to form groups. The heuristics aim to achieve two goals: higher utilization of the system resources and better services to users with shorter response times. To achieve better system utilization, the primary job of a group should coschedule with matched jobs as much as possible in both time and space dimensions. For example, in Figure 9, the right group has better utilization

than the left one since the more "area" of the primary job in right group is coscheduled with matched jobs than that of the left group. To shorten response times, the jobs in the head of the waiting queue have higher priority to be coscheduled since they have been waiting for a longer time than jobs after them (jobs in waiting queue are in FCFS order).

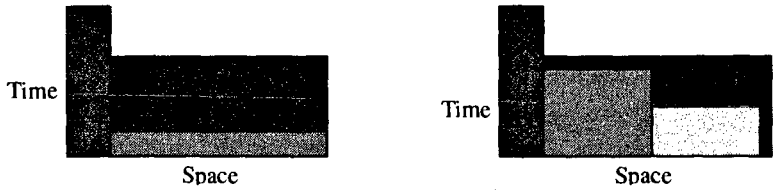


Figure 9. A group with poor utilization (Left) A group with better utilization (Right)

The primary job can be the first waiting job or one of the least delayed running jobs (coscheduling implies slowdown, so we try to avoid further slowdown of already severely delayed running jobs). After the primary job has been decided, matched jobs are selected with the following steps (if the primary is the first waiting job, the matched jobs are from the running or other waiting jobs; if the primary is a running job, the matched jobs are from waiting jobs):

1. Pre-selection: if a job and the primary job do not slowdown each other severely (less than a threshold), the job is selected as a candidate for matched jobs
2. Sort: for candidate jobs, sort in increasing order of delay if they are running jobs, keep their original FCFS order if they are waiting jobs (less delayed or first come jobs will be selected for coscheduling firstly).
3. Block: divide the sorted candidate jobs into blocks and sort the jobs per block in decreasing order of their runtimes.
4. First block: a "window" with the same node requirement as the primary job "slides" over the first block (Figure 10). Each time, the set of jobs within the window's range is selected as matched jobs (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.

5. 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.
6. Purify: a matched job which slowdowns the primary job most but does not contribute to the utilization gain should be removed from the group.
7. Fairness check: a group which causes any other jobs to be delayed severely (more than a threshold) compared to their $R_{est,i}$ is discarded. Groups kept in Step 4 are tested for fairness in decreasing order of utilization gain until a group passes the check.

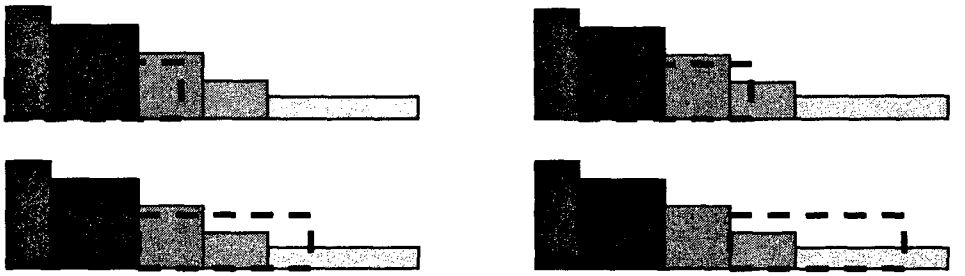


Figure 10. A "window" slides over the first block to search for a set of matched jobs.

4.7. 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 4.2, 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 11. Figure 11 (left) shows the resource requirements of Job 1, 2 and 3 when no coscheduling scheme is applied (there is one and only one process of a job per node), i.e. the runtime does not have any slowdown. Figure 11 (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 time T_g, 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 time T_1 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

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

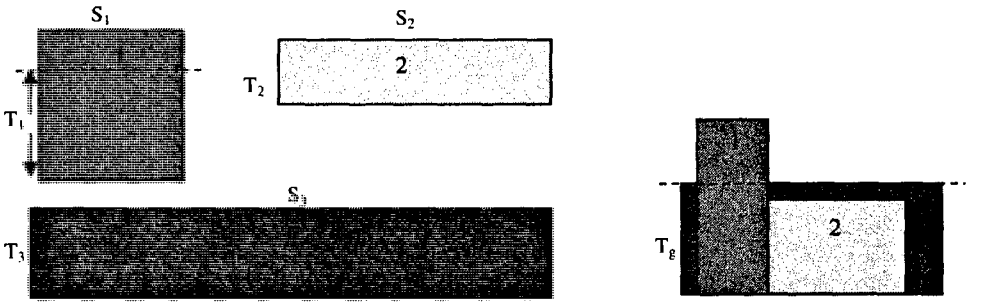


Figure 11. Node usage of Job 1 to Job 3 with space sharing (left) and under coscheduling (right).

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 (the slowdown includes both self slowdown, if possible, and coscheduling slowdown, Section 4.10). There are Q time periods t_q ($1 \le q \le Q$) during the coscheduling of the jobs between changes in the coscheduling status (jobs terminating). $Q \le N$ 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 Fq matched jobs running and the indexes of the matched jobs are $x_1, x_2, x_3 \ldots x_{Fq}$. SL_q ($1 \le q \le Q$) is the maximum slowdown of Job A when A coschedules with the matched F_q jobs running in time period q. Then, $SL_q = \max\{SL_{A,x1}, SL_{A,x2}, \ldots SL_{Ax,Fq}\}$ and the sum of all time periods t_{total} is $t_{total} = \Sigma_{(1 \le q \le Q)} t_q$.

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

$$U_{gain} = (\Sigma_{(1 \le q \le Q)} (\Sigma_{(i \text{ for all } [x1,x2, \ldots xFq])} (t_q*S_i/SL_{i,A}) + t_q*S_A/SL_q) - t_{total}*S_A/PPN_A) /$$
$$(t_{total}*S_A / PPN_A) \quad (6)$$

4.8. Incorporation into Scojo-PECT

Scojo-PECT [7] provides a job scheduler framework. Scojo-PECT employs preemption to support scheduling of shorter jobs even in the presence of long-running ones. 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 [31] but subsequently imposes the constraint that preempted jobs are later restarted on the same resources as migration is not possible without checkpointing. To make preemption to disk affordable and avoid jobs being delayed because of problems with gaining access to their resources again, Scojo-PECT employs coarse-grain time slices and preempts all running jobs. Jobs are divided into 3 types (short, medium and long) based on their runtimes, and scheduled in different time slices/virtual machines with a time slice per job type (a time slice is taken as a 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 [30]. In the context of this thesis, 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 [7].

The basic framework remains to be Scojo-PECT, and G-LOMARC-TS as a job scheduling algorithm is applied per virtual machine/time slice/job type. 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 4.2, the estimated response times and the slack factor define the constraints.

Scojo-PECT is an event-driven simulation scheduler. In total, 4 kinds of events are defined: job-submission, job-termination, slice-begin and slice-end. The happening of each event means the change of system status, e.g. job-submission event means a new job has come and joined the waiting queue; job- termination event means a job has finished running and the resources occupied by the job are emptied out; slice-begin event means a new time slice starts and the jobs with the specific type are resumed/scheduled to run; slice-end event means a slice uses up its time share and all current running jobs are preempted. G-LOMARC-TS is applied to resume/schedule jobs to run after one of the job-submission, job-termination and slice-begin events happens.

4.9. Workload Modeling

For the evaluation of our scheduler, we use the Lublin-Feitelson statistical workload model [18] 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). The Lublin-Feitelson workload model was derived from statistical evaluation of 3 real-life workload traces. The 3 traces came from the following 3 parallel systems:

- San-Diego Supercomputer Center Intel Paragon machine which has 416 nodes (SDSC)
- 1024-node Connection Machine CM-5 installed at Los-Alamos National Lab (LANL)
- 100-node IBM SP2 machine at the Swedish Royal Institute of Technology in Stockholm (KTH)

All the traces contain tens of thousands of jobs and cover months of system activities. The Lublin-Feitelson model generalizes the workload generation to the point that different machine sizes can be chosen. However, the 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 machine 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 [4]. Our goal is to create a workload with a similar machine load 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 (in our simulation, each node has 4 cores totally) or leaving the process number unchanged, depending on the modeled self slowdown (see Section 4.10 for definitions of SL_{crs} and SL_{nos}). 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.
- 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 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 specific combinations of jobs' node requirements and runtimes, this is one of the feasible options of adjustment [30].

The detailed modification applied per job is described in the following formula:

$$newSize = \begin{cases} originalSize & originalSize = 1 \ \| \ (SL_{nos} > SELF\_SL\_2 \ \&\& \ SL_{crs} * SL_{nos} > MAX\_SL) \\ originalSize*2/SL_{nos} & oiginalSize > 1 \ \&\& \ SL_{nos} \leq SELF\_SL\_2 \ \&\& \ SL_{crs} * SL_{nos} > MAX\_SL \\ originalSize*4/\ SL_{crs} / SL_{nos} & originalSize > 1 \ \&\& \ SL_{crs} * SL_{nos} \leq MAX\_SL \end{cases} \quad (7)$$

4.10. Slowdown Modeling

Competition on shared resources like the memory, network, disk and the 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 and can differ depending on whether the processes run on different processors (node self slowdown SL_{nos}) or different cores of the same processor (core self slowdown SL_{crs}). By exploiting different numbers of cores/processors per node, jobs' node requirements are varied. For simplification, we include any relative runtime changes due to the changes of the number of nodes in the self slowdown. Specifically, the coscheduling slowdown and self slowdown of Job i are defined as following:

$$SL_{cos,i} = T_{ij} / T_i \quad (8)$$

$$SL_{nos,i} = T_i' / T_i \quad (9)$$

$$SL_{crs,i} = T_i'' / T_i \quad (10)$$

Here, T_i is the runtime of Job i when there is one and only one process of Job i running on each node; T_{ij} is the runtime of Job i when coscheduled with Job j using semi time sharing per node; T_i' is the runtime of Job i when there are two and only two processes of Job i running on two cores of different processors; T_i'' is similar to T_i' and the difference is that the two processes of Job i run on two cores of the same processor. For example, if there are 4 processes of the same job running on 2 dual-core processors (4 cores totally) per node, the slowdown of the job is $SL_{nos} * SL_{crs}$ since both node and core self slowdown are involved per node.

Slowdowns depend on the applications' characteristics in regards to the usage of resources and require proper slowdown metrics. Since slowdown estimation involves the detailed study of multi-core processors and multi-processor nodes, the software used and the interaction among the processes running in each node, the discussion goes beyond the scope of this thesis, we chose to directly model slowdowns based on available experimental data.

Benchmark configuration	32P/16N vs. 32P/32N	64P/32N vs. 64P/64N	128P/64N vs. 128P/128N	256P/128N vs. 256P/256N
Diffusion for 24^3 problems	1.31	1.46	1.3	1.42
Transport for 24^3 problems	1.05	1.19	1.00	1.16
Diffusion for 48^3 problems	1.61	1.51	1.48	1.57
Transport for 48^3 problems	1.29	1.15	1.09	1.21

Table 2. SL_{crs} calculated for the PARTISN benchmark from data in [4].

	IS	EP	FT	CG	LU	BT*	MG	SP*
16P/8N vs. 16P/16N multi-core	1.37	1.05	1.27	1.04	1.08	1.11	1.22	1.47
16P/8N vs. 16P/16N multi-processor	1.35	1.05	1.16	0.99	0.99	1.00	1.01	1.01

Table 3. SL_{crs} and SL_{nos} for NAS benchmarks from [28].

We took data from [4] 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 processors (T_{single}) and

dual-core processors (T_{dual}), we obtain the slowdown as $SL_{crs} = 2 * T_{dual} / T_{single}$. The calculated slowdowns are shown in Table 2 (showing only machine sizes relevant to our simulation).

In [28], we also obtained data for SL_{crs} 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_{nos} shown in Table 3. Because the data for SL_{crs} is similar to Table 2, we consider the SL_{nos} from Table 3 to be generally valid.

For data in regards to SL_{cos}, we refer to [38] 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). The tests were run on 8, 32, and 64 nodes.

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

Table 4. Model for distribution of slowdown.

From the above discussion, we obtain the distribution model of slowdown shown in Table 4. To simplify the modeling and the slowdown calculation, processes from the same job are currently allocated to cores of different processors per node, while processes from different jobs are allocated to the cores of the same processor. This captures the most frequent cases that processes of the same job run better on different processors rather than on the cores of the same processor (future extensions may consider all possible allocations) because, just as mentioned before, processes of the same job usually have similar resource requirements and cores of the same processor share more resources (e.g. the cache, memory access path) than cores of different processors. If there are N_{core} processes from the same job, they occupy all cores in a node. Figure 12 shows 6 possible allocations in a node in G-LOMARC-TS (the rectangle represents the node, big ellipse the processor, small ellipse the core. A core with a colour means it is occupied by a process; a core with no colour means idle. Different colours mean that the processes are from different jobs.)

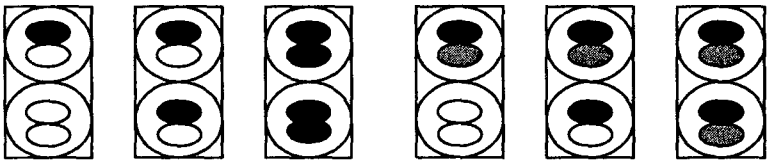


Figure 12. All possible allocations in a node.

5. Experiments and Results Analysis

5.1. Experimental Set-up

We perform the evaluation via discrete event simulation with the workload model described in Section 4.9. Each test with the Lublin-Feitelson workload model is run with 3 random workloads and results are averaged. The cluster used in the simulation has two dual-core processors (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-Feitelson parameters (since our scheduler currently involves 5% overhead[1], 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 (the original estimation is made based on FCFS simulation with space sharing at job submission), we applied an adjustment by a factor of 0.75 to reflect that the estimates do not consider the benefits from non-type slice backfilling and coscheduling and jobs therefore run on average faster than estimated.

W1 (normal load)	$\alpha = 10.33 \rightarrow$ Load = 10.6
W2 (high load)	$\alpha = 9.83 \rightarrow$ Load = 13
Machine size M	128
Short jobs N_S	64%
Medium jobs N_M	19.5% (54% of Medium and Long)
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%
Serial jobs	24%
Power-of-two size among parallel jobs	75%

Table 5. Workload characteristics.

The parameters of Scojo-PECT are set to 30% relative time share for medium jobs and 70%

[1] 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.

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 ≤ 10 minutes, as medium if their runtime is ≤ 3 hours, and as long otherwise.

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

- SSP: Standard space sharing (only one job per node with 1, 2 or 4 processes, depending on the self slowdown)

$$PPN = \begin{cases} 1 & \text{job size} = 1 \;\|\; (SL_{nos} > SELF\_SL\_2 \;\&\&\; SL_{crs} * SL_{nos} > MAX\_SL) \\ 2 & \text{job size} > 1 \;\&\&\; SL_{nos} \le SELF\_SL\_2 \;\&\&\; SL_{crs} * SL_{nos} > MAX\_SL \\ 4 & \text{job size} > 1 \;\&\&\; SL_{crs} * SL_{nos} \le MAX\_SL \end{cases} \quad (11)$$

- GLTS: full group and time/space sharing G-LOMARC-TS
- 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

MAX_SL	1.25	Maximum slowdown that a job can tolerate
SELF_SL_2	1.12	Maximum tolerable 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_EXCEED_PRIM	0.125	If X is the sum of the node requirement of all matched jobs in a group and Y is that of the primary job, X – Y ≤ MATCHED_EXCEED_PRIM * Y
RUNNING_RPIM_NUM	8	Number of running jobs which are considered as primary jobs
MAX_MATCHED_LONGER_PRIM	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

Table 6. Scheduler parameters and values used in the experiments.

All scheduling approaches use conservative backfilling to support prediction. Table 6 shows

all scheduler parameters used in our experiments. We used the following metrics for comparison:

- Average bounded relative response time[2] (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 (Section 4.2)

5.2. General Performance Results

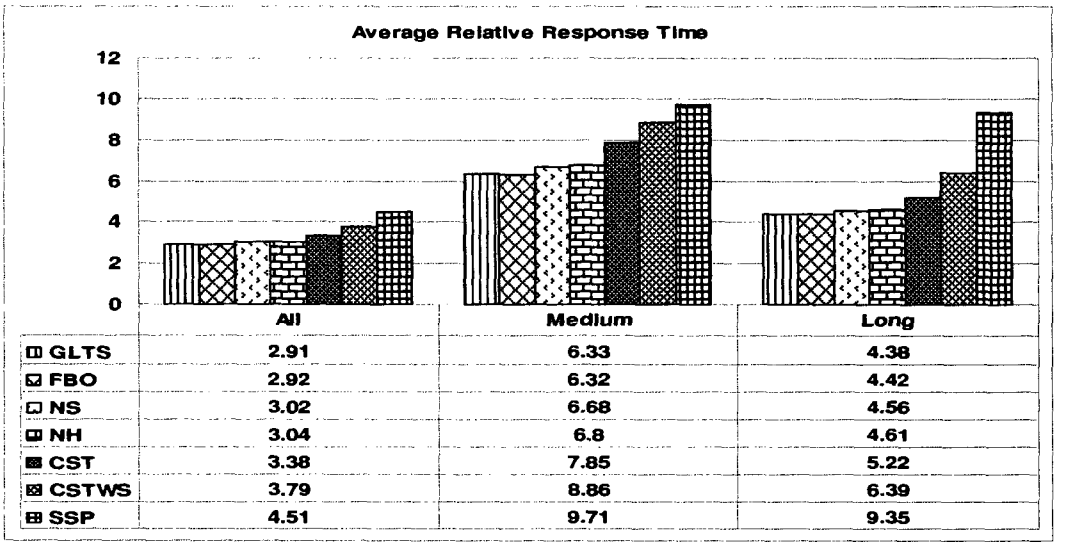
The performance results (measured in RR) for G-LOMARC-TS compared to space sharing (SSP) and matchmaking for only two jobs (CST and CSTWS) are shown in Figure 13. The results show that the full algorithm of G-LOMARC-TS (GLTS) compared to SSP performs by 34.9% better for medium jobs, by 53.2% for long jobs, and 35.5% for all jobs. Note that this means long jobs benefit more. The reason may be that long jobs run a longer time than medium jobs in groups and long-job groups are less frequently disbanded and created than medium-job groups due to their longer runtimes.

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 (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. Here, the improvements are almost the same for long and medium jobs because both CST and CSTWS apply coscheduling with semi time sharing for these two types of jobs.

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

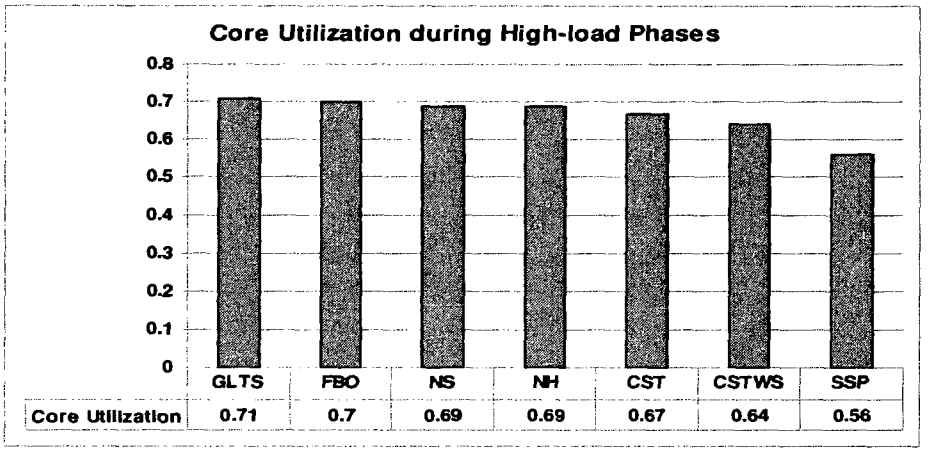
[2] The bounded relative response time is often called bounded slowdown. We avoid this term to avoid confusion with the slowdown due to resource contention.

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.



	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

Figure 13. 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

Figure 14. Core utilizations during high-load phases for different schedulers and G-LOMARC-TS variants with Workload W1.

Figure 14 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. We also found that, just as discussed in Section 4.2, the overall utilization is the same for all scheduling policies since the jobs do not queue-up for any of them.

5.3. Impact of Different Heuristics and Machine Load

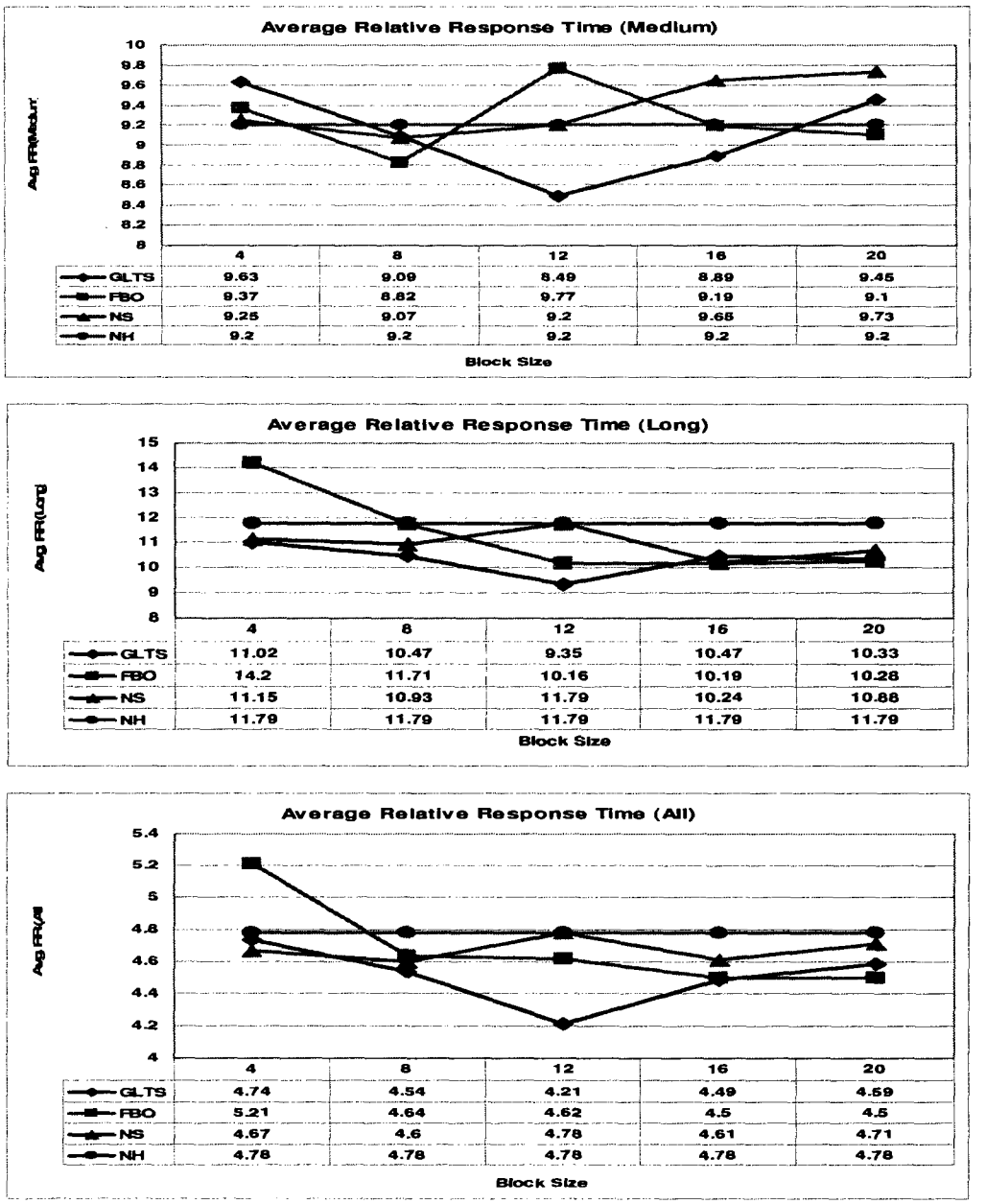


Figure 15. Average relative response time for G-LOMARC-TS and different variants with different block sizes under Workload W2.

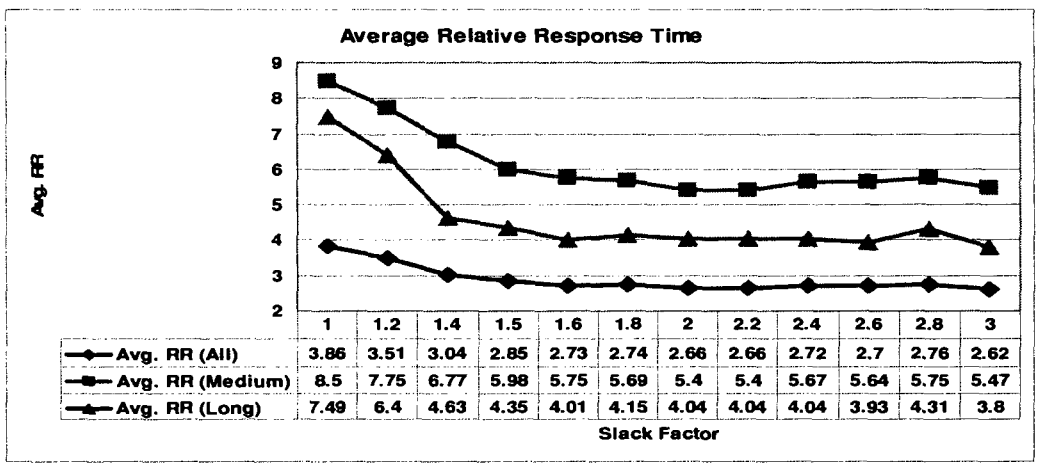
Figure 13 also includes results of several variants of G-LOMARC-TS. However, we do not see much impact from using the full matchmaking algorithm (GLTS) vs. simplifications which only match within the first block (FBO), do not sort (NS), or only selecting the first suitable group (NH). The reason is that 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 15 show that GLTS achieves the best results with optimum block 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 (because, most probably, GLTS finds a best match in the first block). 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 group 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 normal Workload W1. This is due to the fact that the waiting queues become longer and more matching candidates are available.

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.

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 16. 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.



	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

Figure 16. Average relative response time for Workload W1 with different slack factors F_{slack}.

6. Summary and Conclusion

We have presented the G-LOMARC-TS scheduling algorithm which incorporates both space sharing and semi time sharing on clusters with multi-core nodes. With space sharing, a job runs exclusively on a set of nodes with different numbers of processes allocated per node according to the self slowdown of the job; with semi time sharing, jobs with complementary resource characteristics are combined to be allocated to the same set of nodes with different processes on different cores per node. The decision of choosing space or semi time sharing is made according to the machine load, job type and utilization gain. Three forms of groups are supported: 1) the first waiting job coschedules with one or multiple other waiting jobs, 2) the first waiting job coschedules with one or multiple running jobs, 3) a running job coschedules with one or multiple waiting jobs (including the first waiting one). Specifically, the thesis has following contributions:

- Both space and semi time sharing are supported and the selection between them is made based on the machine load, job's type (jobs are divided into 3 types: short, medium and long according to their runtimes, Section 4.8) and utilization gain (Section 4.7). Both sharing schemes involve full and partial usage of cores per node.
- Groups of waiting and/or running jobs are matched while considering the resource contention among processes from the jobs per node.
- Serials of heuristics are applied when trying to create groups since it is an NP-complete problem of forming groups among multiple waiting and/or running jobs.
- Serial jobs are handled specially since there may be bursts of them.
- Only space sharing is applied to schedule short jobs (runtimes are less than a threshold) since short jobs finish soon and it is not worth the effort of coscheduling with semi time sharing in groups.
- A metric is used to select an optimal group which measures how many nodes are saved during a time period due to improved core utilization. This, subsequently, benefits all jobs globally.
- A group itself is optimized by removing jobs which slowdown others most but provide no contribution to the utilization gain.

- No more than 2 jobs are scheduled/coscheduled per node for ease of handling.
- Fairness check is done to avoid serious delay on individual jobs and implemented by allowing a maximum slack factor comparing to the original estimated response time obtained upon job submission.
- There are 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.
- Incorporating other job scheduling techniques such as conservative backfilling (jobs can move ahead in the waiting queue if they do not delay other waiting jobs originally in front of them) to improve system utilization and make runtime estimation possible.

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. The experiments 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 heavier workloads than space sharing per virtual machine. The results also demonstrate that group matchmaking contributes significantly to the improvements.

7. Future Work

One of the tasks for future work is to create a slowdown model which predicts the resource contention effects among processes per node and estimates slowdowns. We need to investigate in detail the behaviors of multi-core processors and multi-processor nodes. Also, we should study the contention effects of processes with different resource characteristics per node. With all these information, the slowdown model should have the ability to estimate the slowdown of one job imposed by another one when they are coscheduled to the same set of nodes with semi time sharing.

We can also extend our work by considering different allocation schemes per node. Specifically, we can include the option of allocating processes from the same job to the cores of the same processor and processes from the different jobs to the cores of different processors. This may benefit if the processes of a job have a great number of shared data since cache sharing is much faster than intro-node communication. As a result, except for the allocations in Figure 12, we can add the following possibilities shown in Figure 17.

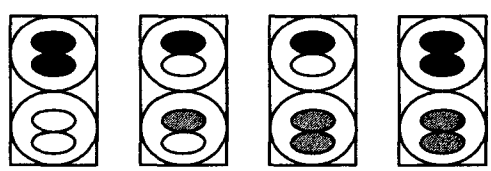


Figure 17. Extended possible allocations in a node.

In this thesis, there are only about 2.5 jobs per group on average because the heuristics used do not purposely select jobs with large size as the primary job in a group. As a result, not many matched jobs can be coscheduled due to the limited size of the primary job. In future work, we can include this heuristic to make primary job larger and then more matched jobs may be added to a group. Consequently, the average number of jobs per group may be increased.

References

[1] S. Agarwal, G.S. Choi, C.R. Das, A.B. Yoo, S. Nagar, Co-ordinated coscheduling in time-sharing clusters through a generic framework, Proc. of the IEEE Intl. Conf. on Cluster Comp., 2003, pp. 84-91.

[2] A.C. Arpaci-Dusseau, D. Culler, Implicit Co-Scheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. ACM Trans. Compu. Sys., Aug. 2001,pp. 283-331.

[3] N. Bansal, M.H. Balter, Analysis of SRPT Scheduling: Investigating Unfairness, ACM SIGMETRICS Performance Evaluation Review, 2001, pp. 279-290.

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

[5] L. Chai, Q. Gao, D. K. Panda, Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System, CCGRID, IEEE Inter. Symp. 2007, pp. 471-478

[6] S.P. Dandamudi, T.K. Thyagaraj, A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems, High Performance Comp. Proc. 4th Intl. Conf., 1997, pp. 218-223.

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

[8] H. Franke, J. Jann, J. Moreira, P. Pattnaik, M. Jette, An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In Proc. 1999 ACM/IEEE Supercomputing Conf., Portland, Nov. 1999.

[9] D.G. Feitelson, L. Rudolph, Gang Scheduling Performance Benefits for Fine-Grain Synchronization. Journal of Parallel and Distributed Computing, Dec 1992, pp. 306–318.

[10] E. Frachtenberg, D. G. Feitelson, F. Petrini, J. Fernandez, Adaptive Parallel Job Scheduling with Flexible Coscheduling, Parallel & Distributed Systems, IEEE Trans., 2005, pp. 1066–1077.

[11] F. Gine, F. Solsona, P. Hernandea, E. Luque, Adjusting Time Slices to Apply Coscheduling Techniques in a Non-dedicated NOW, Euro-Par 2002 Parallel Proc., 2002, pp. 234-239.

[12] F. Gine, F. Solsona, P. Hernandez, E. Luque, Cooperating Coscheduling in a Non-dedicated Cluster, Euro-Par 2003 Parallel Proc., 2003, pp. 212-217.

[13] F. Gine, F. Solsona, M. Hanzich, P. Hernandez, E. Luque, Cooperating Coscheduling: A Coscheduling Proposal Aimed at Non-Dedicated Heterogeneous NOWs, Journal of Computer Science and Technology, Vol. 22, Sep. 2007.

[14] A. Gupta, A. Tucker, S. Urushibara, The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In SIGMETRICS Conf. Measurement and Modeling of Comput. Syst., May 1991, pp. 120–132.

[15] M. Hanzich, F. Gine, P. Hernandez, F. Solsona, E. Luque, A Space and Time Sharing Scheduling Approach for PVM Non-dedicated Clusters, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2005, pp. 379-387.

[16] M. Hanzich, F. Gine, P. Hernandez, F. Solsona, E. Luque, CISNE: A New Integral Approach for Scheduling Parallel Applications on Non-dedicated Clusters, Euro-Par 2005 Parallel Processing, 2005, pp. 220–230.

[17] A. Hori, T. Yokota, Y. Ishikawa, S. Sakai, H. Konaka, M. Maeda, T. Tomokiyo, J. Nolte, H. Matsuoka, K. Okamoto, H. Hirono, Time Space Sharing Scheduling and architectural support, Job Scheduling Strategies for Parallel Processing(JSSPP), 1995, pp. 92-105.

[18] U. Lublin, D.G. Feitelson, The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs, Journal of Parallel and Distributed Computing, 2003, pp. 1105-1122.

[19] A. Moursy, R. Garg, D.H. Albonesi, S. Dwarkadas, Compatible Phase Co-Scheduling on A CMP of Multi-Threaded Processors, Parallel & Distributed Proc. Sys., 20th IEEE Intl., 2006.

[20] T. Moscibroda, O. Mutlu, Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. Proc. Of 16 th USENIX Security Symp., Boston, 2007.

[21] J.K. Ousterhout, Scheduling Techniques for Concurrent Systems, In 3rd Intl. Conf. Distributed Comput. Syst. (ICDCS), Oct 1982, pp. 22–30.

[22] F. Petrini, W. Feng, Buffered coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems, Parallel & Distributed Proc. Sys., 14th IEEE Intl., May 2000, pp. 439-444.

[23] G. Sabin, G. Kochhar, P. Sadayappan, Job Fairness in Non-Preemptive Job Scheduling, Proc. Intern. Conf. on Parallel Proc. IEEE, 2004.

[24] G. Sabin, V. Sahasrabudhe, On Fairness in Distributed Job Scheduling Across Multiple Sites, CLUSTER IEEE, 2004.

[25] G. Sabin, P. Sadayappan, Unfairness Metrics for Space-Sharing Parallel Job Schedulers, Job Scheduling Strategies for Parallel Processing, Springer, Vol. 3834, 2005.

[26] A. Snavely, D.M. Tullsen, G. Voelker, Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor, Proc. of the 2002 ACM SIGMETRICS Intl. Conf. on Measurement & Modeling of Computer Systems, Jun. 2002.

[27] A.C. Sodan, L. Lan, LOMARC — Lookahead Matchmaking for Multi-resource Coscheduling, Job Scheduling Strategies for Parallel Processing (JSSPP), 2005, pp. 288-315.

[28] A.C. Sodan, G. Gupta, A. Deshmeh, X. Zeng, Benefits of Semi Time Sharing and Trading Space vs. Time in Computational Grids, Technical Report 08-020, University of Windsor, Computer Science, May 2008.

[29] A.C. Sodan, L. Lan, LOMARC Lookahead Matchmaking for Multiresource Coscheduling on Hyperthreaded processors, Parallel and Distributed Systems, IEEE Trans., 2006, pp.1360-1375.

[30] A.C. Sodan, Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability—First Experiences, Job Scheduling Strategies for Parallel Processing (JSSPP), 2009.

[31] A.C. Sodan, Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey, Concurrency & Computation: Practice & Experience, 17(15), Dec. 2005, pp. 1725-1781.

[32] G. Stiehr, R.D. Chamberlain, Improving cluster utilization through intelligent processor sharing, Parallel & Distributed Proc. Sys. 20th IEEE Intl., Apr. 2006

[33] D.M. Tullsen, A. Snavely, Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, Internat. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[34] G. Utrera, J. Corbalán, J. Labarta, Scheduling of MPI Applications Self-co-scheduling, Euro-Par 2004 Parallel Processing, 2004, pp. 238-245.

[35] J. Weinberg, A. Snavely, Symbiotic Space-Sharing on SDSC's DataStar System, Job Scheduling Strategies for Parallel Processing (JSSPP), 2007, Vol. 4376.

[36] A.B. Yoo, M. A. Jette, An Efficient and Scalable Coscheduling Technique for Large Symmetric

Multiprocessor Clusters, Job Scheduling Strategies for Parallel Processing (JSSPP), 2001, pp. 21-40.

[37] J.L. Yu, D. Azougagh, J.S. Kim, S.R. Maeng, PROC Process ReOrdering-Based Coscheduling on Workstation Clusters, Parallel & Distributed Proc. Sys.,19th IEEE Intl., Apr. 2005, pp. 50-50.

[38] X. Zeng, J. Shi, X. Cao, A.C. Sodan, Grid Scheduling with ATOP-Grid under Time Sharing, CoreGrid Workshop on Grid Middleware (in conjunction with ISC), Dresden, Springer, Jun. 2007.

[39] Y. Zhang, A. Sivasubramaniam, Scheduling Best-effort and Real-time Pipelined Applications , on Time-Shared Clusters, Proc. of the 13th annual ACM Sys. on parallel algorithms and architectures (SPAA), July 2001.

[40] B.B Zhou, R.P. Brent, On the Development of an Efficient Coscheduling System, Job Scheduling Strategies for Parallel Procesing (JSSPP), Springer, 2001, pp. 103-115.

Vita Auctoris

NAME:	Xijie Zeng
PLACE OF BIRTH:	Zhejiang, P.R. China
YEAR OF BIRTH:	1979
EDUCATION:	The 1th High School, PingYang, Zhejiang, China 1994 – 1997
	Beijing University of Technology, Beijing, China 1997 – 2002
	University of Windsor, Windsor, Ontario, Canada 2006 – 2009