LOMARC—Lookahead Matchmaking for Multi-Resource Coscheduling

Angela C. Sodan and Lei Lan
University of Windsor, Canada
acsodan@cs.uwindsor.ca, lan_lei@hotmail.com

Abstract

Job scheduling typically focuses on the CPU with little work existing to include I/O or memory. Time-shared execution provides the chance to hide I/O and long-communication latencies though potentially creating a memory conflict. We consider two different cases: standard local CPU scheduling and coscheduling on hyperthreaded CPUs. The latter supports coscheduling without any context switches and provides additional options for CPU-internal resource sharing. We present an approach that includes all possible resources into the schedule optimization and improves utilization by coscheduling two jobs if feasible. Our LOMARC approach partially reorders the queue by lookahead to increase the potential to find good matches. In simulations based on the workload model of [Lublin2003], we have obtained improvements of about 50% in both response times and relative bounded response times on hyperthreaded CPUs (i.e. cut times by half) and of about 25% on standard CPUs for our LOMARC scheduling approach.

1. Introduction

The primary goal in job scheduling is to provide good response times to users. A secondary goal is to improve utilization. Both objectives may conflict with each other though often improved response times also mean improved (though potentially not optimum) utilization. The relationship between them is typically not well expressed. The best response-time behavior so far has been reported for gang scheduling [Moreira1998, Feitelson1997]. Gang scheduling is a time-sharing approach and means that all processes of the same job are scheduled across nodes at the same time by globally synchronous time slicing [Feitelson1997]. Gang scheduling has shortcomings as regards latency hiding for I/O and long-latency communication. Latency hiding plays an increasingly significant role for the emerging class of data-intensive applications like datamining. Long-latency hiding is important also for potential grid applications. Loosely coordinated coscheduling [Arpaci1996, Sobalvarro1998, Nagar1999, Zhang2000, Sodan2004] (avoiding the globally synchronous execution and enabling to release the CPU if waiting) and relaxed combinations of gang and local CPU scheduling [Silva1999] provide alternatives performing better in this regard. Loosely coordinated coscheduling requires modifications of the communication software and potentially the OS, typically using a spin-blocking approach to release the CPU after some time of waiting and a priority boost to schedule processes that have been waiting for communication. Hyperthreading processors like the Xeon and the new Intel Pentium 4 make it possible to run two applications at the same time without any context switches and without the need to change the communication software. However, the processes compete for CPU-internal and network resources in addition to memory and I/O. Thus, interesting new options for time-shared execution are available but have to be handled carefully. The target architecture considered is a cluster with high-performance network (like Myrinet or Quadrics [Zhou2004]) and user-level communication. In this paper, we only consider single-CPU nodes (hyperthreaded or standard) but our approach would be extendible to multi-way nodes. Considering the possibilities of hyperthreaded CPUs, we limit our coscheduling to a maximum of two jobs, i.e. a multiprogramming level of 2. In the following, we use the term coscheduling in the sense of running multiple jobs together.

The objectives for our own LOMARC job scheduler are:

- Inclusion of all relevant resources (CPU, network, disk, memory)
- Support of standard time-shared execution on standard CPU and coscheduling on hyperthreaded CPUs
- Increase of utilization though keeping basic primary goal of improved response times

- Usage of application characteristics via a-priori knowledge
- Usage of otherwise standard state-of-the-art scheduling approaches (priorities, backfilling etc.)

We address our objectives by the following innovative solutions:

- Optimizing the schedule by matching two applications whenever possible to share resources for high utilization
- Calculating estimates for response-time impact and utilization improvement while considering reordering to match jobs
- Including application characteristics (CPU, network, disk, memory) via an integrated cost model to estimate matchability and slowdowns
- Relaxing the scheduling order and sorting jobs more flexibly by permitting jobs to move ahead in the schedule if they pair well with other jobs though potentially to some extent pushing other jobs backward in the queue

We apply the standard per-job approach for scheduling jobs, i.e. do not attempt any global optimization. The reasons are that global optimization has a high—$O(n^3)$—time complexity and that its benefit is even questionable, considering that the submissions are dynamic and, in standard approaches with priorities, the overall context of jobs changes permanently.

We have tested our approach via an event-based simulation and the workload described in [Lublin2003], comparing it to standard space-shared job scheduling. Our tests include investigations of different heuristics, focusing either on utilization or response-time impact. We present a maximum slowdown model for the cases of resource competition and validate our estimates by practical tests with synthetic programs on a cluster with hyperthreaded CPUs.

2. Related Scheduling Work

Space/time-shared execution of parallel programs has been shown to outperform mere space sharing by providing better response times [Moreira1998]. The typical practically applied approach for time/space sharing is gang scheduling which means globally synchronized execution of parallel programs [Ousterhout1982]. We have shown that with adaptive space allocation, we can obtain even better response times with a lower multiprogramming level [SodanHuang2004]. This has the essential benefit of reducing memory pressure. Furthermore, gang scheduling has shortcomings with respect to the overhead involved and not being able to hide I/O and other long latencies unless the application internally is doing that. Most parallel applications avoid I/O and compute in-core. However, data-intensive applications like datamining are emerging. Several different approaches have been proposed for a loosely coordinated form of coscheduling (implicit and dynamic coscheduling, periodic boost) which is more flexible and can hide latencies. [Arpaci1996, Sobalvarro1998, Nagar1999, Zhang2000]. See also [Sodan2004] for a survey. Most loosely-coordinated coscheduling approaches apply spin-blocking at the waiting side to avoid wasting CPU time if the partner process is not currently scheduled. Furthermore, some form of priority boost is applied at the receiving side for processes that are waiting for communication but are not currently scheduled. These mechanisms are supposed to keep jobs coscheduled if they are in synchrony and drive processes into synchrony if they are not currently coscheduled but communicating with each other. Loosely coordinated coscheduling is, however, in experimental status. One system reflecting some of these findings is Sun MPI [SunMPI2001], though in own experiments on an SMP server, we found that it does not satisfactorily accomplishes coscheduling [SodanHuang2004, SodanRiyadh2002]. To overcome the I/O problems of gang scheduling and the problems of proper coscheduling for applications with high communication intensity, flexible coscheduling with a combination of gang and local CPU scheduling has been proposed [Silva1999, Fracht2003]. The main idea is to keep frequently communicating applications gang scheduled, while relaxing the scheduling toward local CPU scheduling for coarse-grain applications that potentially have I/O or communication with long latencies. The decision can be made dynamically and per node.

One possible approach to schedule jobs with different combinations of I/O-bound and computation-bound jobs in gang scheduling is to reorder the gang-matrix rows to match jobs in the schedule and schedule them together [Wiseman2003]. The benefit of this approach is that it is dynamic, i.e. does not depend on pre-knowledge about characteristics and can accommodate different phases of the programs, e.g. jobs switching between I/O-bound and computation-bound phases. Then, jobs can be paired or not be paired in different phases. However, this approach needs to use the maximum I/O time of different jobs per row and requires a larger number of rows for choice, i.e. a high multiprogramming level. However, a large multiprogramming level is undesirable as regards memory pressure and the probability of actually finding pairs on large machines with potentially many different jobs per row is low. Flexible coscheduling as described above [Silva1999, Fracht2003] overcomes the problem of different jobs in the row behaving differently and the dependence on the maximum per row but still depends on which jobs are randomly allocated to the same nodes as candidates for matching.

Most approaches apply a heuristic on a per-job basis to allocate jobs and determine the schedule. There is little work to perform a more global optimization. One approach optimizes the job ordering during backfilling (instead of using the common first-fit heuristic) to obtain better response times and utilization. A certain lookahead window is applied and the solution found via dynamic programming [Shmueli2003]. Slack-based scheduling [Talby1999] not only considers multiple factors for priority calculation but is more ambitious as regards finding optimum schedules. The approach, in principle, permits free reordering of the whole queue but sets constraints by the slack that represents maximum delays per job. In a practical setting, the approach boils down to a number of different possible heuristics. In this approach, priority-based heuristics performed best and utilization-based ones worst.

For all approaches of job scheduling, memory pressure creates constraints for scheduling which can increase fragmentation and response time significantly [Setia1999, Batat2000]. All of the above consider only one resource (I/O or memory) in addition to the computation. The approach in [Lein1999] can handle several resources, trying to balance the overall resource usage. The approach is applied during backfilling and searches the whole queue to find the best match. In [Cirne2003], an optimal resource allocation in the sense of adapting the size of the job is found by, at the time of submission, simulating different possible job sizes with the current job queue and selecting the optimum.

3. Hyperthreading

Hyperthreading is a special case of simultaneous multithreading [Tullsen1995] with 2 threads (of the same or different applications) running simultaneously, based on the idea of letting multiple threads share the internal CPU resources in each cycle to increase their utilization. This addresses the problem that modern superscalar processors often cannot keep all their resources busy with a single program. The Xeon hyperthreaded physical CPU has only minor extensions (5% die) to support multiple architectural states—the rest of the resources including the L1 data cache and the L2-L3 unified caches are shared [Marr2002]. Hyperthreading is not limited to the Xeon processor but will become widespread with the Intel Pentium 4. However, the effectiveness of Hyperthreading depends on how well a single thread already would utilize the resources of the CPU and to what extent the threads compete for resources—such as integer and floating-point units—or complement each other. Furthermore, the impact of stalls due to insufficient instruction-level parallelism and branch misses is reduced. Another problem is the sharing of the cache which is typically a scarce resource anyway. The impact of this effect depends on the cache behavior of the program. If the working set is large but just fits nicely into the cache (which may mean that the application is cache-optimized), the competition of a second process/thread running on the CPU can severely slow down the program. However, future versions of hyperthreaded CPUs may perform better by increased cache sizes. Applications that sequentially run over a large set of data in a single pass may perform very well because having little locality (this may apply to, e.g., many datamining applications in comparison to, e.g., a matrix multiplications which use the same rows and columns multiple times). If the program has no cache locality (because of irregular accesses or poor implementation), the effects of longer-latency memory accesses can even be mitigated. Though, memory can equally well create an additional problem if the machine architecture does not provide sufficient memory bandwidth to support two processes as this is often the case [BehrSodan2001]. Parallel applications typically use different data subsets per process/thread and thus compete for the cache. In addition, scientific applications often use more floating-point operations and are already well optimized for them and, thus, can keep the floating-point resources busy with a single thread [Leng2002]. [Magro2002] comes to the conclusion that scientific applications typically show less improvement than business applications (10%-30% vs. 60%). Symbiotic scheduling [Tullsen00] and MASA [Nakajima2002] monitor resource conflicts among running jobs on single-CPU simultaneous multithreading processors and coschedule the jobs that have the least resource contention.

Hyperthreading provides a different option of coscheduling by running multiple applications together on the same physical CPU. This saves overhead for context switches and coordination.

Especially applications that are dominated by floating-point operations can run well together with applications that are dominated by integer operations [Nakajima2002]. Though, the threads have to share the network, with communication not only creating network contention but also memory-access contention. In [Leng2002], the communication effects were studied and, for communication-intensive benchmarks, a degradation in performance was observed. In [Nakajima2002], an approach is presented to set affinity to certain physical or logical CPUs at user level. This would make it possible to extend our approach to run on dual SMP nodes. Furthermore, the involved modification of the OS-internal CPU scheduling can be used to switch hyperthreading dynamically on and off (i.e. switch from multithreading mode MT to single-threaded mode ST). This can be done by using the priviledged (OS) instruction hlt (HALT).

4. The Slowdown Estimation and Empirical Evaluation

4.1 The Slowdown Estimation

For the following discussion, we first need to define our view of slowdown. Note that we always assume two jobs being coscheduled.

Definition *individual-execution-slowdown*: The factor in execution time by which an application A runs slower in joint execution with another application B ($T_{A,B}$) than it would run on its own (T_A), i.e. $sl_{A,B} = T_{A,B} / T_{A.}$.

Note that this definition is different from the slowdown definition in loosely coordinated coscheduling such as implicit coscheduling [Arpaci1996] which bases on jobs normally running twice as long in joint time-shared execution. Thus, the slowdown is the relative factor beyond that, i.e. $T_{A,B} / (2\ T_A)$ if $T_A \leq T_B$. For example, if two jobs with equal runtime together run 3 times as long, the slowdown is considered to be 3/2 = 1.5. Since our concern is increasing utilization, this view is not appropriate for us.

Previous research [Magro2002, Leng2002] has investigated the performance on hyperthreaded SMP nodes and/or cluster for applications as a whole. Thus, no detailing into computation and synchronization/communication cost was done and no I/O was considered. Below we present a slightly more detailed model which estimates the maximum slowdown. We split execution time into the fraction of computation time f_{comp}, the fraction of communication time f_{comm}, and the fraction of I/O time f_{io}. For simplification, we assume that $f_{comp} + f_{comm} + f_{io} = 1$, i.e. we currently do not consider any application-internal latency hiding. For applications with many short communications, we may actually attribute most of the communication time (similar to [Figueira2001]) as computation time because most of the time ($f_{comm,\mathrm{O,Lmcopy}}$) is spent on the CPU for setting up the communication, copying to and from buffers, polling to wait on communication, and copying between host and NI (network interface) memory (because typically being buffered and handled via Programmed I/O—PIO). Long communication involves little CPU time because employing Direct Memory Access—DMA—and zero-copy communication [Zhou2004]. Similarly, I/O spends a certain amount of time $f_{io,OS}$ in OS handling—especially buffer copying—on the CPU. We basically assume I/O is to the local disk—if I/O goes to an I/O server, the message-passing part ($f_{io,comm}$) has to be attributed to the network. Thus,

- $f_{CPU} = f_{comp} + f_{comm,O,Lmcopy} + f_{io,OS}$
- $f_{network} = f_{comm} - f_{comm,O,Lmcopyn} + f_{io,comm}$
- $f_{disk} = f_{io} - f_{io,OS} - f_{io,comm}$

with f_{CPU} being the time on the CPU, $f_{network}$ the time on the network, and f_{disk} the time on the disk.

In the perfect case, applications would exploit different resources all the time but typically $T_{A,B} \geq T_A$.

Disk, network, and CPU usages do not conflict with each other. In the general case, applications use all three resources though in different shares. Race conditions may apply and, in the worst case, the applications are using the same resources at the same time, and we therefore have to estimate competition on resources. Cost estimates have to consider worst case behavior per node because the probability for the worst case to happen increases with the number of nodes, converging to a probability of 1. The potential for conflicts is described below for the different resources.

Communication: Two jobs may communicate at the same time: the communication will be serialized on the NI and in the DMA. On different nodes, communications may interleave in different order, leading to delays for both applications. Since according to our measurements, non L2/L3 cache integer operations have little slowdown, we can ignore additional CPU time from added polling time. Thus, we estimate the slowdown as

$$sl_{A,B,network\cdot} = min\ \{f_{network,A}, f_{network,B}\} * 2\ / f_{network,A}$$

Hyperthreaded CPUs: they compete for floating-point and integer CPU-internal resources and for the cache. The former serializes instructions, the latter creates additional cache misses. The exact resource competition depends on how much instruction parallelism is available per application and which execution resources are needed at any time vs. the available resources in the CPU. In [Magro2002], the major difference made is between integer- and floating-point-dominated applications. However, in own measurements, we found a somewhat more complex relationship. As regards the cache, we found that often cache-miss latencies can be hidden within the application or among applications. Thus, coscheduling two applications with cache conflict does not necessarily reduce performance significantly more than if there are no conflicts. Furthermore, the sum of the cache-space needs does not linearly translate into cache misses because caches are not perfectly LRU (Least Recently Used) but n-way direct (the Xeon L2/L3 caches are 8-way) caches that may lead to replacements even if the working set still fits into the cache. We estimate

$$sl_{A,B,CPU} = (f_{A,B,competing} * 2 + f_{A,B,different}) * \min\{f_{CPU,A}, f_{CPU,B}\} + sl_{A,B,mem}) / f_{CPU,A}$$

with $f_{A,B,competing}$ being the fraction of the code competing for CPU-internal resources and $f_{A,B,different} = 1 - f_{A,B,competing}$ the fraction using different resources. Detailed modeling would require an advanced cache/CPU cost model and a detailed application model (access patterns, instructions mixture) which goes beyond the scope of this paper. Similar arguments apply to $sl_{A,B,mem}$ which expresses the slowdown from paging if the two applications do not fit into memory together. We therefore have obtained upper-bound parameters empirically (see below). Note that a slowdown of 2 corresponds to time-sharing on a standard CPU and that any slowdown > 2 means a decrease in utilization.

<u>I/O:</u> the system calls for I/O will be partially serialized, may interfere with each other by going to different tracks (and therefore adding seek times), and compete for buffer space. However, the different I/O calls may also provide potential for OS-internal optimization or overlapping each other on the disk. The details depend on the OS. We make the assumption that the same serialization of cost applies as for the other cost components, i.e.

$$sl_{A,B,disk} = \min\{f_{disk,A}, f_{disk,B}\} * 2 / f_{disk,A}$$

This leads to the following overall maximum slowdown:

$$sl_{A,B} = sl_{B,A} = (f_{A,B,competing} * 2 + f_{A,B,different}) * \min\{f_{CPU,A}, f_{CPU,B}\} + sl_{A,B,mem} + \min\{f_{network,A}, f_{network,B}\} * 2 + \min\{f_{disk,A}, f_{disk,B}\} * 2$$

Note that the slowdown for *A* and *B* is the same (since we count the shared parts) and that the maximum slowdown according to the above formula is 2 as long as no memory conflicts are involved. The slowdown is the lower, the more different the characteristics of the two applications are, i.e. the smaller the shared parts on the different resources.

Information about application characteristics can be obtained by monitoring shortened sample runs or by monitoring normal application runs and keeping the information for future runs in performance databases [Gibbons1997]. A tool like Paradyn [Miller1995] may be used to obtain the standard characteristics f_{comp}, f_{comm}, f_{io}. Vtune [VtuneIntel] can obtain performance counters for CPU-internal usage and measure, for example, retired floating-point operations and cache misses to obtain estimates about CPU-internal resource usage and conflicts.

Considering the discussion above, we can now compare our coscheduling on hyperthreaded CPUs to loosely coordinated coscheduling. The latter can hide I/O latency though I/O intensive applications can significantly disturb the coordinated execution of intensively communicating jobs (and cause process switches and delays). Thus, both types of jobs should not be coscheduled. However, this negative effect does not exist on hyperthreaded processors because both applications can continue to execute at any time. For loosely coordinated coscheduling of communication and/or computation-dominated jobs, the best results obtained so far are about a factor of 2.4 slowdown, and it is not even sure whether these results generalize. Thus, the benefits are more limited. We only coschedule jobs if we can obtain a benefit, i.e. a slowdown below a $sl_{limit} \leq 2$. As a benefit of loosely coordinated coscheduling, it is less sensitive to the cache though the spin-block also in a negative cache impact (process switches on standard CPUs invalidate the whole cache) [Sodan2004].

Above, we have made the simplification not to consider application-internal latency hiding. Such consideration is, however, possible. We only have to make sure to recognize that no external latency-hiding potential is available anymore for the corresponding fractions of the code. We can simply mark these fractions as the combination of the typically two resource types. An estimation on the safe side, then, is to count the whole combined fraction for each of the corresponding resource types when estimating conflicts. Latency hiding (and improved resource usage) is still possible for such applications if matching with an application which is dominant in the third resource type.

4.2 Empirical Evaluation of Slowdowns

We have tested slowdowns with synthetic applications on a cluster with Intel Xeon processor and Myrinet interconnect, running MPICH-GM with user-level MPI communication. L2/L3 cache size is 512k and memory size per node 512 Mbyte. The operating system is Linux 2.4. In all measurements, we use f_{comp}, f_{comm}, and f_{io} due to our current lack of low-level monitoring tools that could reveal the CPU, network, and disk fraction. We checked that the single-process performance is almost identical for the CPU set to MT or ST mode.

We first investigate hyperthreaded CPU behavior and run applications dominating in either float or integer calculations, dominating in complex multiplication/division or simpler add instructions, running totally in L1 cache or using some or much of the L2/L3 cache. The code sequences are simple and easily fit into the cache. The summary of results can be seen in Table 1. As far as L2/L3 usage is involved, we have modeled an access patterns that runs over the same data structure serially per iteration (except to totally irregular accesses, this is the worst situation because under LRU all data would be repeatedly replaced if not fitting totally into the cache). Each computation step accesses 4 close-by elements (as would be the case if calculating the stencil in a mesh computation). Our results are consistent with other research as far as available. In [Magro2002], scientific applications benefited between 10% and 30% by running each with two threads on a hyperthreaded CPU. However, even performance on a dual SMP was not optimal. Thus, translating the hyperthreading improvement to the relative best-possible threaded performance, the slowdowns according to our definition were approximately 1.4 which is not worse than the up to 30% improvement measured for business applications. [Leng2002] shows slowdowns up to 3, including communication, for cache-intensive applications. Since the tests were done by increasing the number of processes per application, however, also the speedup behavior changed (speedup curves typically flatten with larger number of processes) and the results therefore appear to be too negative.

Table 1. Slowdown for different types of computation. + means application uses add operations, * means it uses mult operations; the number indicates the size of the data in L2/L3 cache.

	float+ 0	int + 0	float+ 40k	int + 40k	float+ 400k	int + 400k	float* 0	int* 0	float* 80k	int* 80k	float* 400k	int* 400k
2x same	2.1	1.07	1	1.3	1.5	1.3	1.6	1.4	1.1	1.1	1.7	1.2
float and same type int	1.1 / 1						1.8 / 1.4		1.2 / 1.1		1.7 / 1.4	

The results of our tests show that float applications with simple add operations and no L2/L3 cache data usage provide the poorest performance as the resources are apparently well utilized. Integer applications generally tend to coschedule better because utilizing the available resources less intensively. Applications with L2/L3 cache usage run relatively slower but coschedule in several cases better than the versions with no L2/L3 usage. Thus, cache misses appear to be partially hidden within the thread or among the threads and, for detailed estimation, an integrated model would be required.

In Table 2, we show results from running applications together with a) different mixtures of communication and computation, and b) different communication granularity. In all cases, $f_{io}=0$. Runtime for each application on its own is 60 sec. In all cases, the applications are run on 4 nodes and are loosely synchronous, communicating with all 3 other neighbors, sending to them and receiving from them in each communication phase. The computations are of type "int + 0" to focus on the effects of CPU vs. network. Note that the short communication is spending a significant amount of time on the CPU via PIO (with integer operations), whereas the long communication employs DMA and zero copy in a rendezvous protocol. C_{size} is the number of bytes per communication. All runtimes are in seconds. Communication cost results into 13.3 µsec for a message with 200 bytes and into 120 µsec for a message with 18,000 bytes. In all cases, the actual slowdown is lower than the estimated maximum slowdown. As can be seen from the table, the slowdown is different for each application if running coarse- and fine-grain communication together. The application with the finer communication (smaller and more communications) suffers more. The explanation is that if the communications interleave, the finer-grain communications are stretched more, adding idling time to this application. If applications are slowed down to different extent, it would be important to make sure that enough non-competitive time is left for the application with the larger slowdown to catch up with communication. Thus, additional conditions for the matching may be necessary to ensure that $f_{network,A}+f_{network,B} \leq 1$ (not currently considered).

Table 2. Runtimes and slowdowns for coscheduling two applications with different mixtures of computation and communication (and different communication granularities). The left number represents the row application, the right number the column application. Since the applications finish at different times, we have added a projected time *Pt*, representing the runtime if the other application would have continued to run. *Sl_m* is the measured slowdown, *$sl_{e,max}$* is the estimated slowdown, with the number in parenthesis being the estimate if there would be no slowdown in the CPU part.

	f_{comm}=0.4, C_{size}=200	f_{comm}=0.4, C_{size}=18,000	f_{comm}=0.6, C_{size}=200	f_{comm}=0.6, C_{size}=18,000
f_{comm}=0.4, C_{size}=200	68 / 68 sl_m: 1.1 / 1.1 $sl_{e,max}$: 2 (1.4)	70.5 (Pt=72.5) / 62 sl_m: 1.2 / 1 $sl_{e,max}$: 2 (1.4)	68 / 69 sl_m: 1.1 / 1.1 $sl_{e,max}$: 1.8 (1.4)	75 (Pt=77) / 64 sl_m: 1.3 / 1.1 $sl_{e,max}$: 1.8 (1.4)
f_{comm}=0.4, C_{size}=18000		65 / 65 sl_m: 1.1 / 1.1 $sl_{e,max}$: 2 (1.4)	62 / 73 (Pt=76) sl_m: 1 / 1.3 $sl_{e,max}$: 1.8 (1.4)	64 / 64 sl_m: 1.1 / 1.1 $sl_{e,max}$: 1.8 (1.4)
f_{comm}=0.6, C_{size}=200			72 / 72 sl_m: 1.2 / 1.2 $sl_{e,max}$: 2 (1.6)	79 (Pt=84) / 64 sl_m: 1.4 / 1/1 $sl_{e,max}$: 2 (1.6)
f_{comm}=0.6, C_{size}=18,000				67 / 67 sl_m: 1.1 / 1.1 $sl_{e,max}$: 2 (1.6)

Finally, we show in Table 3 our results of testing different classes of applications together. In this case, we found no difference in whether the computation is on integers or floating points. The I/O is repeatedly reading a 60 Mbyte file sequentially in 1k blocks from the local disk. The communicating application is running a standard pingpong test. Note that the combinations using two communication or two I/O intensive applications are stress-tests only—LOMARC would not normally coschedule such applications.

Table 3. Slowdown if running different classes of applications together.

	f_{comm}=1, C_{size}=200	f_{comm}=1, C_{size}=18k	f_{comp}=1, int + 0	f_{io}=1
f_{comm}=1, C_{size}=200	1.4 / 1.4	1.7 / 1.1	1 / 1	1 / 1.3
f_{comm}=1, C_{size}=18k		1.3 / 1.3	1 / 1	1.1 / 1.2
f_{comp}=1, int + 0			1.1/ 1.1	1 / 1.2
f_{io}=1				1.2 / 1.2 to 2.3 / 2.3

The results show that there is little negative impact if the job classes are different. Surprisingly, the I/O is slowed down by communication (and not vice versa as in loosely coordinated coscheduling). An explanation is that both the communication and the I/O still involve significant CPU time. The interference of two I/O applications is very indeterministic though in most of the cases in the range of *sl = 1.2*.

Finally, we have studied the effect of paging. Using the same type of application as for cache measurements, we have compared the effect of running two applications with 400 Mbyte and 267 Mbyte memory usage each. The slowdown is 2.5 in the former and 2.2 in the latter case. Thus, the difference is not very high though the difference in conflict is significant.

In summary, our measurements show that there are no unexpected superlinear slowdowns and that conversely the slowdowns actually measured are in many cases much lower than our maximum estimate (though the slowdowns may increase with larger numbers of nodes). Thus, by using these estimates, we make very conservative assumptions for the evaluation of our scheduling algorithm.

5. The Look-Ahead Scheduling Algorithm

5.1 The General Algorithm

We apply a standard job-scheduling algorithm with the following features

- Usage of priorities, classifying the jobs into short, medium, and long and allocating priorities according to these classes; usage of aging to prevent starvation.
- First-fit during allocation of jobs onto nodes
- Flexible and dynamic allocation of nodes (no fixed and contiguous partitions required)
- Backfilling (EASY backfilling)

We basically keep short response times as the primary schedule-optimization objective and exploit utilization as far as it does not contradict good response times. However, we propose different heuristics, mainly aiming at either optimization for response times (as it would be meaningful during the day) or optimization for utilization (as it would be meaningful during the night). Memory consumption currently only plays the role of a constraint.

The key special features in our LOMARC scheduling approach are:

- Estimating the utilization gain
- Estimating the impact on the response times
- Allocating jobs to free nodes by themselves if the accumulated node requests in the queue ≤ the available nodes by 20% (machine is weakly loaded)
- Finding a possible best match for the next job subject to scheduling among
  - The remaining jobs in the waiting queue
  - The running jobs

This means that LOMARC never coschedules jobs if the machine is weakly loaded, i.e. there are empty nodes to run the job. We classify jobs into CPU-bound, disk-bound, and network-bound, according to which of f_{CPU}, f_{disk}, or $f_{network}$ dominates. Only medium and long jobs are considered for coscheduling.—short ones are not worth the effort.

LOMARC can schedule either on standard or hyperthreaded CPUs with the following scheme:

- On a standard CPU, we only schedule CPU-bound and disk-bound jobs together. Only they can benefit as regards CPU utilization in this case.
- On a hyperthreaded CPU, more options exist to coschedule jobs. We consider joint execution of CPU-bound and CPU-bound jobs, CPU-bound and network-bound jobs, and network-bound and I/O-bound jobs in addition to CPU-bound and disk-bound jobs.

Thus, LOMARC does not depend on any special coscheduling software (gang or implicit coscheduling). However, LOMARC depends on the option to share the network [Sodan2004]. Such sharing is, however, provided by the widespread standard native GM communication library for Myrinet and the MPICH and LAM MPI implementations that build on top of GM [Zhou2004].

Figure 1 shows pseudo code of the abstracted LOMARC algorithm. Figure 2 and Figure 3 graphically demonstrate the matchmaking.

Our LOMARC algorithm depends on knowing the characteristics of the applications as regards the fractions of time on CPU, network, and disk and making correct upper bound estimations for slowdowns. We assume the applications to be occasionally monitored (we have accompanying research work running on this topic). If the estimates

```
while (! waiting_queue.is_empty ()) {                // run over all jobs in queue as long as can
   current_job = waiting_queue.first;                // be scheduler
   while (current_job.size <= freenodes.size) {      // enough space for job
      if (current_job.is_medium_or_long_job () )     // try find a match for the job among
             match = find_match (current_job);       // remaining jobs in waiting queue
      allocate_nodes (current_job);
      if (match != null)
            coallocate_nodes (current_job, match);   // coallocate match on same nodes
      if (end_of_queue) return ();
        else current_job = waiting_queue.first;
      }
   if (current_job.is_medium_or_long_job)            // current job won't fit on free nodes
     { match = find_match_among_running (current_job)         // co-schedule with running job
      if (match != null)                             // find best match among running
             coallocate_nodes (match, current_job);  // allocate current job on same nodes
      }
      if (match == null)                             // current job does not match any job
         break;                                      // current job cannot be scheduled now;
   }                                                 // continue with backfilling
}                                                    // end of loop running over queue
backfill ();                                         // try to backfill jobs onto free nodes
                                                     // (applying same matching as above)
```

Figure 1. Abstracted LOMARC scheduling algorithm as invoked upon job-termination or submission.

are severely wrong in a negative sense, one application may be preempted and its execution be completed when the other one is finished [Niko2002]. Shorter overall job runtimes than estimated, however, do not hurt at all as we can try to find a new match if one job finishes.

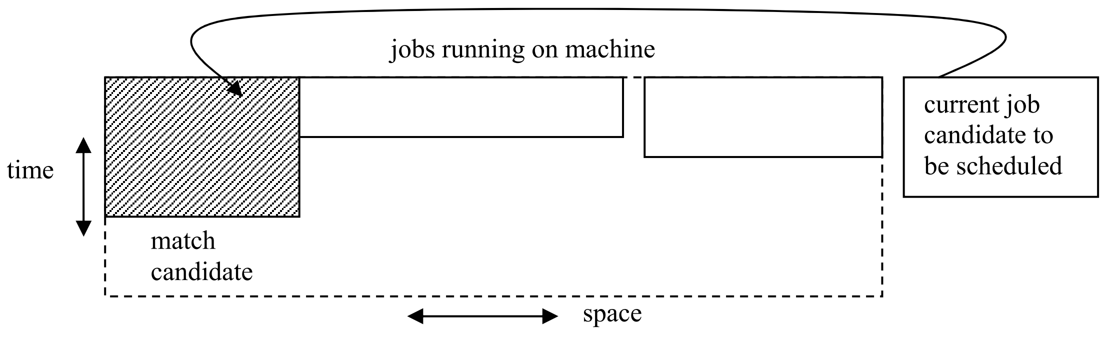


Figure 2. Finding best match among currently running jobs.

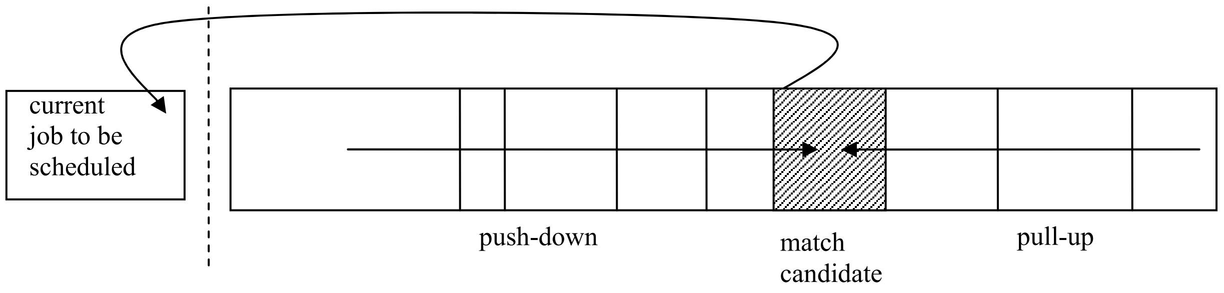


Figure 3. Reordering the job queue if finding a match in the waiting queue.

5.2 The Utilization-Gain and Response-Time-Impact Calculation

Figure 4 shows the search for the best match among all jobs in the waiting queue (if searching there) and the definition of matchable jobs. We first check whether job classes can be matched (e.g. whether their requirements fit). Furthermore, we estimate the slowdown according to our description above. If the slowdown is less than a certain threshold sl_{limit} (MAX_SLOWDOWN), the job becomes a candidate for matching. Different heuristics can be applied as explained below. Either response-time impact and utilization gain can be estimated.

The calculation of the response-time impact does not consider any detailed packing, i.e. does not calculate any actual schedule. The reason is that the packing anyway is subject to change under dynamic submission with priorities. Furthermore, the complexity of incorporating such calculation is high—backfilling has $O(n^2)$ time complexity and, if trying all jobs in the waiting queue to find the optimum, complexity increases to $O(n^3)$. Thus, we simply assume that a perfect packing would be possible (by taking work = runtime * size for each job and adding the corresponding work up for all jobs) and determine all delays on the basis of this simple heuristic. A future improvement might be to calculate exact order for the first few jobs in the queue and apply the heuristic estimate for the rest.

As regards utilization, a detailed utilization metric would have to consider the maximum capacity of hyperthreaded CPUs, disk, and network and their utilization by each application (making detailed resource and application models necessary). Therefore, instead of absolute utilization, we consider the relative utilization improvement on the basis of the scheduled applications.

Definition *Relative Utilization Gain*: We consider the overlap in time where the two jobs run together and calculate how much faster the jobs run if coscheduled than they would run if scheduled individually. We have the following two options: to consider a timeless metric ($U_{gain,2}$) or to include the shared (overlap) runtime that is affected by the utilization change ($U_{gain,1}$). This leads to the following two formulas:

$U_{gain,1} = (min(S_A, S_B) * (2/sl_{A,B}-1) - |S_A-S_B| * (1-1/sl_{A,B})) * (min(T_A,T_B) / max(T_A,T_B)) / max(S_A, S_B)$

$U_{gain,2} = (min(S_A, S_B) * (2/sl_{A,B}-1) - |S_A-S_B| * (1-1/sl_{A,B})) / max(S_A, S_B)$

with *S* being job size.

```
find_match (job) {
    maxprofit = 0;
    match = null;
    for each_job_in_queue (match_cand) {
      if (matchable (job, match_cand)){
        slowdown_cand = slowdown (job, match_cand);              // determine slowdown
        if (slowdown_cand <= MAX_SLOWDOWN) {                     //set limit for slowdown
        switch (heuristic) {
          case 1: profit = utilization_gain_1 (job, match_cand);        // utilization gain1
          case 2: profit =utilization_gain_2(job,match_cand);           // utilization gain 2
          case 3: profit = response_time (job, match_cand, slowdown_cand);  // response times
         }
         if profit > maxprofit                      // keep best match
              {maxprofit = profit;
               match = match_cand;}
        }
      } }
    return match;
}

matchable(jobi, jobj){
if (jobi.memory + jobj.memory <=1)
    if (jobi is CPU intensive && jobj is CPU intensive) return true;
      else (if jobi.type !=jobj.type ) return true;
 return false;
}
```

Figure 4. Finding best match in waiting queue and definition of matchable jobs.

As regards relative response times, the impact from reordering the queue can be estimated in the following way:

- Jobs in front of the job that is matched and thus moved ahead get delayed: For them, we calculate an estimate of the impact by the sum of all relative delays. We call these jobs ***push-down*** jobs.
- Jobs behind the job that is matched get scheduled earlier, assuming that the match decreases the joint runtime of the two jobs vs. running them on their own: For these jobs, we calculate an estimate of the impact by the sum of all relative improvements. We call these jobs ***pull-up*** jobs.

In both cases, we include a prediction about future job submissions and the impact of these jobs on response times. We do a one-level prediction, calculating new job submisssions in the time interval which we estimate for the execution of the jobs that are currently in the queue. To do so, we use parameters (average work) from the workload model. We simplify the calculation of relative response times by taking them relative from the current time on.

See Figure 5 for the details of the algorithm.

The complexity of our algorithm is $O(n^2)$. However, the worst case for searching through all jobs in the queue—O (nlgn)—is always met if we look for the optimum match. To check whether we can reduce cost, we also incorporate a simplified version in our experiments that takes the first match.

```
// calculates overall response-time impact, in increase/decrease relative to normal response time
response_time (jobi, jobj, slowdown) {
        pairruntime = min (jobi.runtime, jobj.runtime) * (slowdown-1) +
                      max (jobi.runtime, jobj.runtime);
        pairsize = max (jobi.size, jobj.size);
        improvement = jobi.runtime*jobi.size / n_nodes;
        delay = (pairruntime * pairsize – jobi.runtime*jobi.size) / n_nodes;
        response_decrease = jobj.runtime*jobj.size / n_nodes – delay;
        response_increase = delay / responsetime;

       //estimate delay for push-down jobs
        for (all push_down_jobs (jobn)) {
            response_time += jobn.runtime * jobn.size / n_nodes;
            response_increase += delay / response_time;
           }
        // response time improvement for job be job being moved
        response_time += jobj.runtime* jobj.size / n_nodes;
        response_decrease = (response_time – jobj.runtime* slowdown * jobj.size / n_nodes)
                                                                        / response_time;
       // estimate improvement for pull-up jobs
        for (all_pull_up_jobs) {
          response_time += jobn.runtime * jobn.size / n_nodes;
          response_decrease += improvement /  response_time;
          }
       for (future_arrival_short_jobs(jobn)) {
          response_time  = jobn.runtime * jobn.size  / n_nodes;
          response_increase+= delay / response_time;
          }
        for (future_arrival_med_or_long jobs (jobn)) {
            response_time  = jobn.runtime * jobn.size / n_nodes;
            response_decrease+= improvement / response_time;
          }
     return (response_decrease – response_increase) / (number(push_down_job) – number(pull-up_jobs);
     }
```

Figure 5. Pseudo code for abstracted calculation of utilization gain and response-time impact.

6. Experimental Results

Our experiments are based on an event simulation with parameter settings and workload modeling as described below. The machine modeled is a cluster with 128 single-CPU nodes.

6.1 Metrics and Workloads

We use the following metrics to evaluate the performance of our LOMARC scheduling algorithm:

- Average response times
- Average relative bounded response times: response time in relation to runtime time, bounded by a 60 sec minimum runtime to avoid overly high impact of very small jobs
- Utilization: percentage of used-nodes time over the makespan ; i.e., ratio of the accumulated used nodes and the product of makespan T and number of nodes P
- Utilization efficiency: if coscheduling, also considers positive improvements by increasing the utilization per CPU, indirectly reflected by a shortened makespan:

$$E = \frac{\sum_i p_i t_i}{PT}$$

with p_i and t_i being size and runtime per job

- Makespan: the runtime of the whole job batch

We have used the model in [Lublin2003] for the workload generation. This model is a complex statistical workload description, considering job sizes, job runtimes, and job interarrival times. The model includes correlations between sizes and runtimes, fractions of sequential jobs, fractions of power-of-two sizes, and differing interarrival times according to day/night-cycles. All numbers are generated in logarithmic space. A two-stage uniform distribution is used for job sizes (including probabilities for serial and power-of-two job sizes), a hyper-Gamma distribution for job runtimes, and two Gamma distributions for interarrival times (one for peak times and one for the overall daily cycle). The parameters of the model are extracted from three traces of supercomputing centers and propose a generalization from the three test cases. The nice feature of this generalized model is that it can be adapted to different machine sizes and, thus, be applied to our machine size of 128 nodes. We have modeled 8,000 jobs.

Furthermore, we have modified the original workload by shortening the job interarrival times, determined by the α parameter of the Gamma distribution. Workload 1 is the original workload, Workload 2 and Workload 3 have smaller α parameters as shown in Table 4. The table also shows the resulting load value $Load = (r-n)/(P*a)$ with r being the mean runtime, n the mean job size, and a the mean job interarrival time [Lublin2003].

Table 4. Workloads modeled.

	Workload 1	Workload 2	Workload 3
α	10.23	9.83	8.83
Load	10.6	13	21

To the best of our knowledge, there do not exist any studies on the distribution of the application's resource-usage characteristics as regards CPU, network, and disk. We model the following mixtures

- M1: 40% CPU-bound, 30% network-bound, 30% disk-bound
- M2: 40% CPU-bound, 10% network-bound, 50% disk-bound
- M3: 30% CPU-bound, 50% network-bound, 20% disk-bound

We perform the majority of our tests with the mixture M1 which can be considered the mixture we expect to see on clusters with a share of scientific and datamining applications. We do some comparisons that include M3 as a representation of what might be the conventional mixture and M2 which might be the mixture for clusters specializing on datamining.

Detailed job characteristics are generated randomly, using an equal distribution per value range, according to the following scheme:

- CPU-bound jobs: f_{CPU} in [0.5,0.9), f_{disk} in [0.05,0.4) with $f_{cpu} + f_{disk}$ in [0.6,0.95)
- Disk-bound jobs: f_{disk} in [0.4,0.65), $f_{network}$ in [0.05,0.4) with $f_{disk} + f_{network}$ in [0.5,0.8)
- Network-bound jobs: $f_{network}$ in [0.4,0.65), f_{disk} in [0.05,0.4) with $f_{network} + f_{disk}$ in [0.5,0.8)

As regards the CPU-behavior, we model different probabilities that the CPU-parts of the two applications go well or poorly together, i.e. increase or decrease utilization. We set the probability for the former case to $p=0.33$ and for the latter to $p=0.67$. We assume $sl_{CPU}=1.4$ in the former and $sl_{CPU}=2$ which picks two typical cases from our measurements in Section 4.1. In the latter case, LOMARC does not schedule CPU-bound applications together.

Memory consumption is modeled by random generation for each job in [0.05,1] with 70% of the jobs in [0.05,0.5], 25% in (0.5,0.8), and 5% in [0.8.1]. 1 represents the maximum memory size that is available for applications. This distribution is roughly modeled as an average over existing memory studies as in [Chiang2001]. LOMARC does not coschedule jobs that do not fit into memory together but, for the comparison with other scheduling approaches, we need to model the memory slowdown and set $sl_{mem}=2.5$ according to our measurements in Section 4.2.

6.2 Experiments with LOMARC Scheduler

To evaluate the benefits of our approach, we compare to

- Standard single-job scheduling (mere space sharing PSS)
- Always coscheduling two jobs if running on a hyperthreaded CPU (AC)
- Coscheduling two jobs that are adjacent in the queue if they are a match according to the LOMARC definition (AM) if running on hyperthreaded CPUs

For our LOMARC approach, we test the following variants:

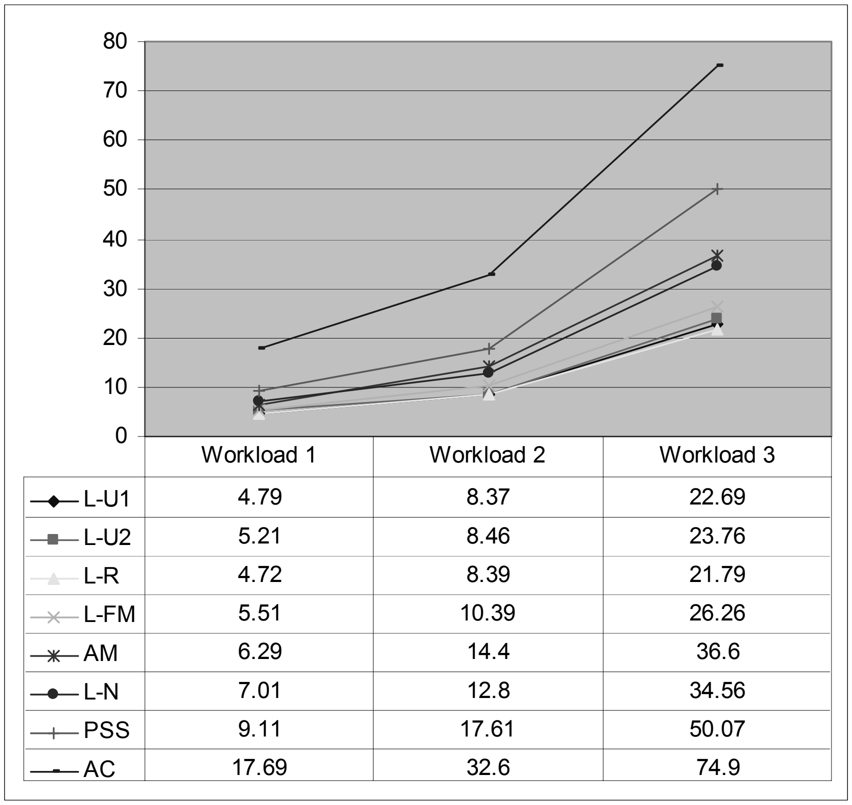
- Scheduling on standard CPUs (L-N) using $U_{gain,1}$
- Optimization with different heuristics on hyperthreaded CPUs: utilization $U_{gain,1}$ (L-U1) and $U_{gain,2}$ (L-U2), response-time impact (L-R), and a variant which selects the first match found (L-FM)

We set the maximum acceptable slowdown. MAX_SLOWDOWN to 1.6. For all approaches, we use priorities and EASY backfilling. We define job classes in the following way: runtimes in [1sec,1min] are classified as short, in (1min,1h] as medium, and in [1h,45h] as long with 45h being the maximum runtime modeled. Aging (to prevent starvation) is based on average waiting time T_{age}. Per each T_{age}., the priority of one job will be boosted to a higher level, so it will take a long job $2T_{age}$ to have the same priority as a short job.

In Figure 6 and Figure 7, we show the results from comparing several LOMARC variants (L-U1 L-U2, L-R, L-FM, and L-N) with PSS, AC, and AM under the 3 different workloads W1, W2, and W3. In all cases, the characteristics mix M1 is used. For all workloads, all LOMARC variants perform clearly better than all other approaches. The arbitrary coscheduling AC is signficantly worse than space sharing PSS and, thus, not a reasonable choice. This demonstates that detailed match considerations are necessary to make coscheduling on a hyperthreaded CPU meaningful.

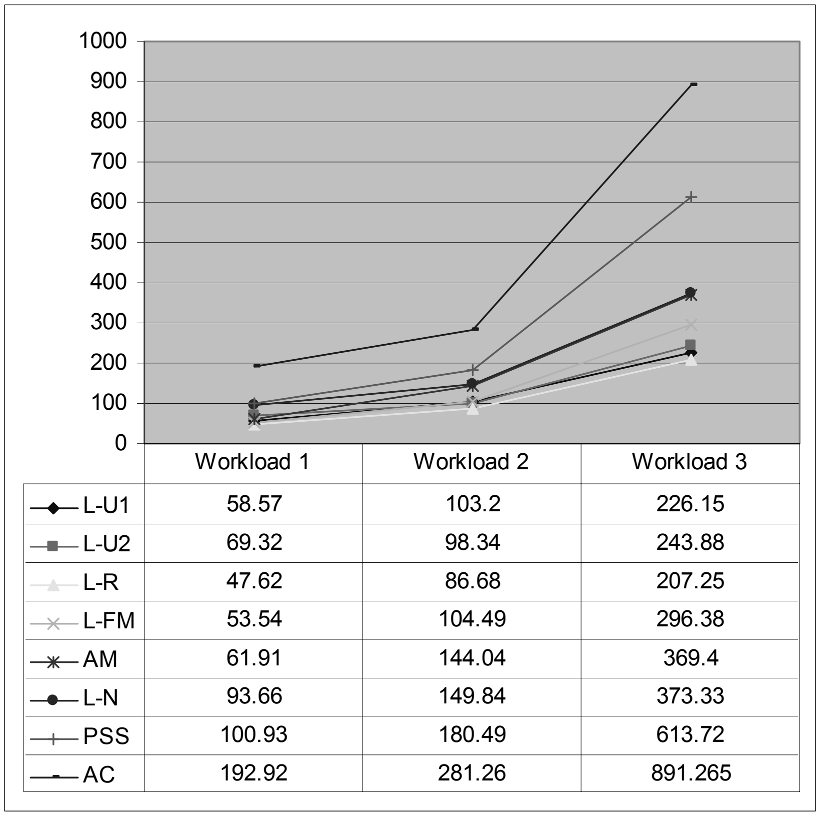
We can see that with the workload becoming heavier, our approaches, L-U1, L-U2, L-R, L-FM and L-N, show more obvious improvement over other approaches in response time, relative bounded response time and effective utilization. The improvement in response time of L-R increases from 48% to 56%, and the improvement in relative bounded response time of L-R increases from 50% to 66% compared to PSS. Thus, response time and relative bounded response time are approximately reduced to half by using our approach.

Comparing our different LOMARC heuristics, all are pretty close to each other as regards response times. However, L-U1 performs slightly better than L-U2. L-U1 provides almost the same results as L-R for all workloads. The differences are more pronounced for the relative bounded response times. L-U2 is again worse than L-U1. Obviously, $U_{gain,1}$ provides the more adequate estmate. L-R is better than L-U1, especially for W1 where it is better by 19% whereas only better by 16% for W2, and by 8% for W3. To perform better as regards relative response times is the expected result for a metric focusing on them. Selecting simply the first match in L-FM is not too much worse if the workload is lighter (W1) but becomes worse than the workload becomes heavier where there are more choices to select the match but ignored by this approach. Response times are by 17% worse than L-R under W3 and relative bounded response times by 24%. Relating the performance to PSS, the improvement in response times of L-FM vs. PSS is 40% for W1, 41%



	Workload 1	Workload 2	Workload 3
◆ L-U1	4.79	8.37	22.69
■ L-U2	5.21	8.46	23.76
▲ L-R	4.72	8.39	21.79
× L-FM	5.51	10.39	26.26
✳ AM	6.29	14.4	36.6
● L-N	7.01	12.8	34.56
+ PSS	9.11	17.61	50.07
- AC	17.69	32.6	74.9

Figure 6. Response times for different scheduling approaches and different workloads.



	Workload 1	Workload 2	Workload 3
◆ L-U1	58.57	103.2	226.15
■ L-U2	69.32	98.34	243.88
▲ L-R	47.62	86.68	207.25
× L-FM	53.54	104.49	296.38
✱ AM	61.91	144.04	369.4
● L-N	93.66	149.84	373.33
+ PSS	100.93	180.49	613.72
- AC	192.92	281.26	891.265

Figure 7. Relative bounded response times for different scheduling approaches and different workloads.

for W2, and 48% for W3. The improvement in relative bounded response times is 47% for W1, 42% for W2, and 52% for W3. Thus, the much simpler heuristic provides still very good results. AM that only matches adjacent jobs in the queue is still doing significantly better than PSS but still significantly worse than L-R and also worse than L-FM, especially for heavier workloads. Considering scheduling on standard CPUs (L-N), LOMARC still provides significant improvements: as regards response times 23% for W1, 27% for W2, and 31% for W3. Relative bounded response times are improved by 7% for W1, 17% for W2, and 40% for W3.

The makespans for Workload 1 are about 10 weeks and are by only 5% improved by L-R vs. PSS. This indicates that there are often not enough jobs to fully utilize the machine. The improvement for Workload 2 is 20% and for Workload 33%.

In Figure 8, we show utilization and utilization efficiency for all approaches. Utilization is almost the same for all approaches and for all approaches improves if the workload becomes heavier (because more options for packing exist). For utilization efficiency, LOMARC shows improvements, especially under heavier workloads, for L-U1, L-U2, and L-R: 8.5% for W1, 19% for W2, and 38% for W3. However, there are no relevant differences between L-U1, L-U2, and L-R. This means using a heuristic which focuses on utilization does not make any difference. L-FM is slightly worse—the improvement is 6% for W1, 15% for W2, and 31% for W3. L-N only accomplishes 2% improvement for L1, 9% for W2, and 18% for W3.

To check how much difference the assumptions about the characteristics mix make, we present response times and relative bounded response times for Workload 1 and M2 and M3 in Figure 9.

For M2 and M3, the relative improvements of L-U1, L-U2, and L-R vs. PSS and AC are similar to M1. However, for M2, L-N now improves upon PSS by 34% in response times and by 38% in relative bounded response times and for M3 it is closer to PSS than under M1. This is consistent with the expectation because in M2 there are more disk-bound jobs that can still be coscheduled with CPU-bound jobs and, in M3, there are fewer of them.

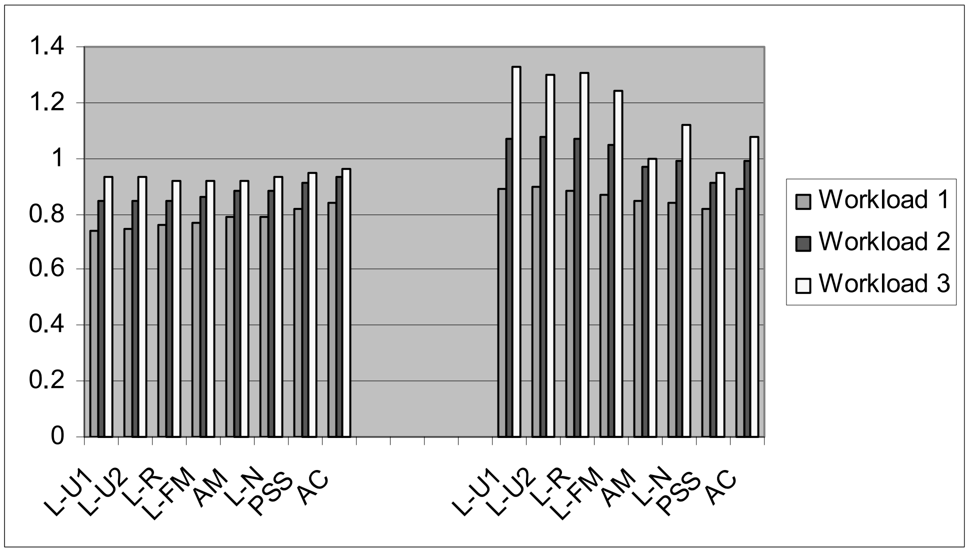
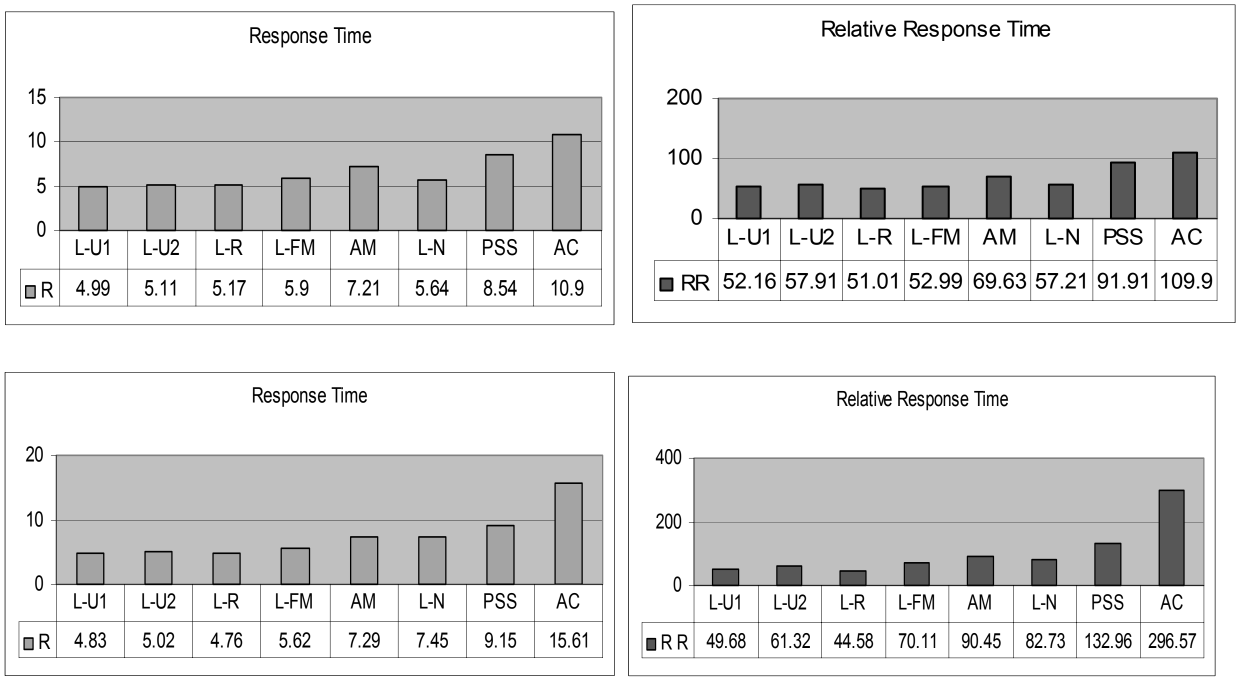


Figure 8. Utilization (left) and utilization efficiency (right) for different scheduling approaches and different workloads.



	L-U1	L-U2	L-R	L-FM	AM	L-N	PSS	AC
▨ R	4.99	5.11	5.17	5.9	7.21	5.64	8.54	10.9

	L-U1	L-U2	L-R	L-FM	AM	L-N	PSS	AC
■ RR	52.16	57.91	51.01	52.99	69.63	57.21	91.91	109.9

	L-U1	L-U2	L-R	L-FM	AM	L-N	PSS	AC
▨ R	4.83	5.02	4.76	5.62	7.29	7.45	9.15	15.61

	L-U1	L-U2	L-R	L-FM	AM	L-N	PSS	AC
■ R R	49.68	61.32	44.58	70.11	90.45	82.73	132.96	296.57

Figure 9. Average response times and average relative bounded response time for M2 (upper row) and M3 (lower row).

Finally, we investigate the detailed behavior of L-U1, L-U2, and L-R (under M1) by looking at the average queue lengths, the number of jobs left in each comparison step for finding a match, and which job in the end is selected. See Table 5. As we can see, after meeting all the constraints, the number of jobs left as candidates to choose from by the different heuristics is relatively small: for W1 between 5 and 7. With this small number of choices, there is not much room for the different heuristics to create different effects. For all heuristics, on average the 3rd match candidate is selected. L-U1 and L-R select the 4th match candidate under Workload W2. For Workload 3, we see a significant difference: L-U1 selects the 6th job and L-R the 4th which is an expected effect as optimizing with a focus on response times should be more reluctant to select a job which has a position further down in the queue. However, the results for response times and relative bounded response times as discussed above do not really confirm this as the actual improvement of L-R is higher for W1.

Table 5. Average queue lengths, average numbers of jobs left under the different constraints, and average job selected for candidates.

		Average Queue Length	Medium or Long Job	SizeB ≤ SizeA	Memory Fit	Matchable	Slowdown ≤ Max	Number Selected
L-U1	Workload1	36	24	10	8	6	5	3
	Workload2	81	45	21	12	10	8	4
	Workload3	213	86	40	20	16	14	6
L-U2	Workload1	40	25	11	8	6	5	3
	Workload2	77	49	25	16	12	11	5
	Workload3	233	85	39	21	16	14	7
L-R	Workload1	36	24	13	11	8	7	3
	Workload2	80	51	31	21	15	14	4
	Workload3	222	87	48	25	20	17	4

7. Summary and Conclusion

We have presented an approach to find matches between two jobs on hyperthreaded and standard CPUs for better resource utilization via coscheduling. The approach partially reorders the queue and searches for the best match while estimating impacts on relative bounded response times and utilization. In simulations, we have shown that our LOMARC scheduler clearly outperforms standard space sharing as regards response times and relative bounded response times by reducing them to about half their original value on hyperthreaded CPUs and to about ¾ on standard CPUs. The heuristic performing best is to estimate the response-time impact when selecting the best match. The improvement is accomplished by an improvement in utilization efficiency from running multiple jobs with complementary resource requirements. Each individual application is unlikely to accomplish the same internally,, especially if the application does not use multithreading per CPU but simply doubles the number of processes. Worth to note, our improvements from LOMARC have been accomplished with quite conservative assumptions about slowdowns.

Future work includes a refined slowdown model, experiments with other simplified heuristics (like making the choice between the first three candidates only or selecting a candidate if it is beyond a certain match threshold), and testing the scheduler with conservative backfilling which may be more sensitive to whether utilization or response-time impact is considered. Furthermore, extension to multi-way nodes is of interest. Then, another choice is to schedule one or multiple applications on the different CPUs per node. For such nodes, applications are more likely to be prepared to use multithreading per node and may already use the network very intensively. Thus, there may be fewer options for coscheduling as regards network usage but also new options in using physical and virtual CPUs.

Acknowledgements

This research was partially supported by NSERC and by CFI (Grant No. 6191) with contributions from OIT and IBM. Thanks to Xuemin Huang for providing a basic version of the job-scheduling simulator.

References

[Arpaci1996] Andrea Dusseau, R. Arpaci and D. E. Culler. Implicit Scheduling: Efficient Distributed Scheduling for Parallel Workloads on Networks of Workstations. Proc. SIGMETRICS Conf. Measurement and Modelling of Computer Systems, Philadelphia/PA, USA, 1996.

[Batat2000] Anat Batat and Dror G. Feitelson. Gang Scheduling with Memory Considerations. Proc. IPDPS, 2000.

[BehrSodan2001] Peter Behr, Samuel Pletner, and Angela C. Sodan. The PowerMANNA Architecture. IEEE Conference on High Performance Computer Architecture (HPCA), Toulouse, France, January 2000, pp. 277-286.

[Chiang2001] S.-H. Chiang and M.K. Vernon. Characteristics of a Large Shared Memory Production Workload. Proc. JSSPP, 2001.

[Cirne2003] W. Cirne and F. Berman. When the Herd is Smart: Aggregate Behavior in the Selection of Job Request. IEEE Trans. on Parallel and Distributed Systems, Vol. 14, No. 2, Feb. 2003.

[Feitelson1997] Feitelson D G. Job Scheduling in Multiprogrammed Parallel Systems, Extended Version. Technical Report, IBM, August 1997, RC 19790 (87657).

[Figueira2001] Silvia M. Figueira and Francine Berman. A Slowdown Model for Applications Executing on Time-Shared Clusters of Workstations. IEEE Transactions on Parallel and Distributed Systems, Vol. 12, No. 6, June 2001.

[Fracht2003] Eitan Frachtenberg, Dror Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. Proc. Int. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.

[Gibbons1997] R.A. Gibbons Historical Application Profiler for Use by Parallel Schedulers. Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), April 1997, Lecture Notes in Computer Science **1291**, Springer Verlag.

[Lein1999] Leinberger W, Karypis G, and Kumar V. Job Scheduling in the Presence of Multiple Resource Requirements. Proc. IEEE/ACM Supercomputing Conf.(SC), Seattle/WA, USA, 1999.

[Leng2002] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. An Empirical Study of Hyper-Threading in High Performance Computing Clusters. Linux HPC Revolution, 2002.

[Lublin2003] U. Lublin and D.G. Feitelson.The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing Nov. 2003, **63**(11):1105-1122.

[Magro2002] Wiliam Magro, Paul Peterson, and Sanjiv Shah. Hyper-Threading Technology: Impact on Compute-Intensive Workloads. Intel Technology Journal Q1, Vol. 6, No. 1, 2002.

[Marr2002] D. Marr D, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M.Upton. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q1, Vol. 6, No. 1, 2002.

[Miller1995] Miller B P, Callaghan M D, Cargille J M, Hollingsworth J K, Irvin R B, Karavanic K L, Kunchithapadam K, and Newhall T. 1995. The Paradyn Parallel Performance Measurement Tools. IEEE Computer, Special issue on performance evaluation tools for parallel and distributed computer systems, Nov. 1995, **28**(11):37-46.

[Moreira1998] Jose E. Moreira, Waiman Chan, Liana L.Fong, Hubertus Franke, and Morris A. Jette. An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. Supercomputing'98, Nov. 1998.

[Nagar1999] Shailabh Nagar, Ajit Banerjee , Anand Sivasubramaniam, and Chita R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. Proc. ACM SPAA. Saint Malo, France, 1999.

[Nakajima2002] Jun Nakajima and Venkatesh Pallipadi. Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling. Proc. USENIX 2nd Workshop on Industrial Experiences with Systems Software, Boston/MA, USA, Dec. 2002.

[Niko2002] D.S. Nikolopoulos and C.D. Polychronopoulos. Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs. Proc. International Parallel and Distributed Processing Symposium (IPDPS), Fort Lauderdale/CA, USA, April 2002.

[Ousterhout1982] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. Proc. 3rd Intl. Conf. Distributed Comp. Systems, 1982, pp. 22-30.

[Setia1999] Sanjeev Setia, Mark Squillante, and Vijay K. Naik. The Impact of Job Memory Requirements on Gang-Scheduling Performance. Performance Evaluation Review, March 1999.

[Shmueli2003] Edi Shmueli and Dror G. Feitelson. Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2003.

[Silva1999] Fabricio Alves Barbosa da Silva and Isaac D. Scherson. Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler. Proc. 11th Symp. On Computer Architecture and High Performance Computing, Natal, Brazil, Sept. 1999.

[Sobalvarro1998] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 1998.

[SodanHuang2004] Angela C. Sodan and Xuemin Huang. Adaptive Time/Space Sharing with SCOJO. Conf. on High Performance Computing Systems (HPCS), Winnipeg/Manitoba, May 2004.

[SodanRiyadh02] Angela C. Sodan and Muhammad Riyadh. Coscheduling of MPI and Adaptive Thread Applications in a Solaris Environment. Proc. IASTED PDCS, Cambridge/MA, USA, Nov. 2002.

[Sodan2004] Loosely Coordinated Coscheduling in the Context of Other Dynamic Job Scheduling Approaches–A Survey. Concurrency&Computation: Practice & Experience. To appear.

[SunMPI2001] SUN HPC ClusterTools 4 Performance Guide. SUN Microsystems, August 2001. Retrieved from http://www.sun.com/products-n-solutions/hardware/docs/Software/.

[Talby1999] Talby D and Feitelson D G. Supporting Priorities and Improving Utilization of the IBM SP2

Scheduler Using Slack-Based Backfilling. Proc. IPPS, 1999.

[Tullsen1995] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. Proc. Ann. Int. Symp. on Computer Architecture (ISCA), June 1995.

[Tullsen00] D. M. Tullsen and A. Snavely. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Nov. 2000.

[VtuneIntel] Vtune Performance Analyzer. Intel Corporation, http://www.intel.com

[Wiseman2003] Y. Wiseman and D. G. Feitelson, Paired Gang Scheduling. IEEE Trans. Parallel & Distributed Systems, 2003.

[Zhang2000] Yanyong Zhang, Anand Sivasubramaniam, Jose Moreira,and Hubertus Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. Proc. Int. Conf. on Supercomputing (ICS), Santa Fe /NM, USA, May 2000.

[Zhou2004] Ying (Joy) Zhou and Angela C. Sodan. Survey of Zero-Copy Optimization in User-level Communication and Adaptive Knowledge-Based Solution. Conf. on High Performance Computing Systems (HPCS), May 2004.