Improvements in Parallel Job Scheduling Using Gang Service

Fabricio Alves Barbosa da Silva[1*]

Isaac D. Scherson[1,2†]

fabricio.silva@lip6.fr, isaac@uci.edu

[1]Laboratoire ASIM, LIP6, Université Pierre et Marie Curie, Paris, France.[‡]

[2]Dept. of Information and Comp. Science, University of California, Irvine CA 92697, U.S.A

Abstract

Gang scheduling has been widely used as a practical solution to the dynamic parallel job scheduling problem. Parallel threads of a single job are scheduled for simultaneous execution on a parallel computer even if the job does not fully utilize all available processors. Non allocated processors go idle for the duration of the time quantum assigned to the threads. In this paper we propose a class of scheduling policies, dubbed Concurrent Gang, that is a generalization of gang-scheduling, and allows for the flexible simultaneous scheduling of multiple parallel jobs, thus improving the space sharing characteristics of gang scheduling. However, all the advantages of gang scheduling such as responsiveness, efficient sharing of resources, ease of programming, etc., are maintained.

1 Introduction

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern multiprocessors parallel computers. It is defined as: "Given the aggregate of all threads of multiple jobs in a parallel system, find a spatial and temporal allocation to execute all tasks efficiently". For the purposes of scheduling, we view a computer as a queueing system. An arriving job may wait for some time, receive the required service, and depart [10]. The time associated with the waiting and service phases is a function of the scheduling algorithm and the workload. Some scheduling algorithms may require that a job wait in a queue until all of its required resources become available (as in variable partitioning), while in others, like time slicing, the arriving job receives service immediately through a processor sharing discipline.

We focus on scheduling based on gang service, namely, a paradigm where all threads of a job in the service stage are grouped into a gang and concurrently scheduled in distinct processors. Reasons to consider gang service are responsiveness [6], efficient sharing of resources[11] and ease of programming. In gang service the threads of a job are supplied with an environment that is very similar to a dedicated machine [11]. It is useful to any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling[8]. One main problem related with gang scheduling is the necessity of multi-context switch across the nodes of the processor, which causes difficulty in scaling[4]. In this paper we propose a class of scheduling policies, dubbed concurrent gang, that is a generalization of gang-scheduling and allows for the flexible simultaneous scheduling of multiple parallel jobs in a scalable manner.

The architectural model we will consider in this paper is a distributed memory processor with three main components:1) Processor/memory modules (Processing Element - PE), 2) An interconnection network that provides point to point communication, and 3) A synchronizer, that synchronizes all components at regular intervals of L time units. This architecture model is very similar to the one defined in the BSP model [18]. We shall see that the synchronizer plays a important role in the scalability of gang service algorithms.

Although it can be used with any programming model, Concurrent Gang is intended primarily to schedule efficiently SPMD jobs. The reason is that the SPMD programming style is by far the most used in parallel programming.

[*] Supported by Capes, Brazilian Government, grant number 1897/95-11.

[†] Supported in part by the Irvine Research Unit in Advanced Computing and NASA under grant #NAG5-3692.

[‡] Prof. Greiner is gratefully ackowledged

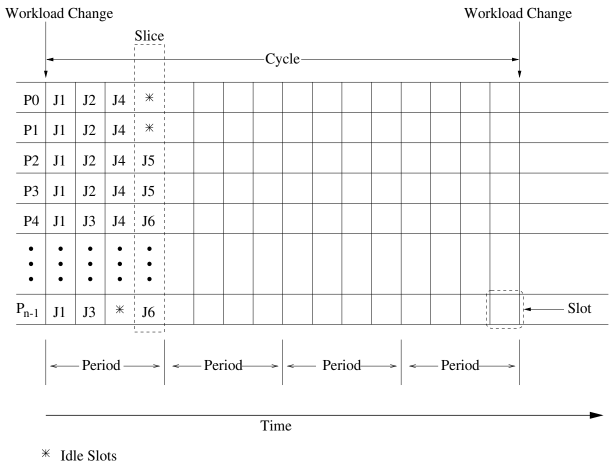

Figure 1. Definition of slice, slot, period and cycle

2 Concurrent Gang

In parallel job scheduling, as the number of processors is grater than one, the time utilization as well as the spatial utilization can be better visualized with the help of a bidimensional diagram dubbed *trace diagram*. One dimension represents processors while the other dimension represents time. Through the trace diagram it is possible to visualize the time utilization of the set of processors given a scheduling algorithm. A similar representation has already been used, for instance, in [16]. One such diagram is illustrated in figure 1

Gang service algorithms are preemptive algorithms. We will be particularly interested in gang service algorithms which are *periodic and preemptive*. Related to periodic preemptive algorithms are the concepts of cycle, slice, period and slot. A *Workload change* occurs at the arrival of a new job, the termination of an existing one, or through the variation of the number of eligible threads of a job to be scheduled. The time between workload changes is defined as a *cycle*. Between workload changes, we may define a period that is a function of the workload and the spatial allocation. The period in the minimum interval of time where all jobs are scheduled at least once. A cycle/period is composed of *slices*; a slice corresponds to a time slice in a partition that includes all processors of the machine. A *slot* is the processors' view of a slice. A Slice is composed of N slots, for a machine with N processors. If a processor has no assigned thread during its slot in a slice, then we have an idle slot. The number of idle slots in a period divided by the total number of slots in the period defines the *Idling Ratio*. Note that workload changes are detected between periods. If, for instance, a job arrives in the middle of a period, corresponding action of allocating the job is only taken by the end of the period.

Referring to figure 2, for the definition of Concurrent Gang we view the parallel machine as composed of a general queue of jobs to be scheduled and a number of servers, each server corresponds to one processor. Each processor may have a set of threads to execute. Scheduling actions are made at two levels: In the case of a workload change, global spatial allocation decisions are made in a front end scheduler, who decides in which portion of the trace diagram the new coming job will run. The switching of local threads in a processor as defined in the trace diagram is made through local schedulers, independently of the front end. In the event of a job arrival, a job termination or a job changing its number of eligible threads (events which define effectively a workload change if we consider moldable jobs) the front end Concurrent Gang Scheduler will :

1 - Update Eligible thread list
2 - Allocate Threads of First Job in General Queue.
3 - While not end of Job Queue
        Allocate all threads of remaining jobs
        using a defined spatial sharing strategy
4 - Run

For rigid jobs, the relevant events which define a workload change are job arrival and job termination.

All processors change context at same time due to a global clock signal coming from a central synchronizer. The local queue positions represents slots in the scheduling trace diagram. The local queue length is the same for all processors and is equal to the number of slices in a period of the schedule. It is worth noting that in the case of a workload change, only the PEs concerned by the modification in the trace diagram are notified.

From the job's perspective, with Concurrent Gang it still has the impression of running in a dedicated machine, as in gang scheduling, except perhaps for some possible reduction in I/O and network bandwidth due to interference from other jobs. Still, the CPU and memory resources required for the job are dedicated.

It is clear that once the first job, if any, in the general queue is allocated, the remaining available resources can be allocated to other eligible threads by using a space sharing strategy. Some possible strategies are first fit and best fit policies which are classical bin-packing policies In first fit, slots are scanned in serial order until a set of slots in a slice with sufficient capacity is found. In best fit, the sets of idle slots in each slice are sorted according to their capacities. The one with the smallest sufficient capacity is chosen. The local queue length is the same for all processors and is equal to the number of slices in a period of the schedule.

In the case of creation of a new thread by a parallel thread, or parallel thread termination, it is up to the local scheduler to inform the front end of the workload change. The front end will then take the appropriate actions depending on the pre-defined space sharing strategy.

A local scheduler in Concurrent Gang is composed of two main parts: the Gang scheduler and the local task scheduler. The Gang Scheduler schedules the next thread indicated in the trace diagram at the arrival of a synchronization signal. The local task scheduler is responsible for scheduling local threads that do not need global coordination and it is similar to a Unix scheduler. The Gang Scheduler has precedence over the local task scheduler.

We may consider two classes of threads in a concurrent gang scheduler: Those that are eligible to be scheduled as a gang with other threads in other processors and those that are not. An example of the first class are threads that compose a job with fine grain synchronization interactions. Second class thread examples are local threads or threads that compose an I/O bound parallel job, for instance. In [13] Lee et al. proved that response time of I/O bound jobs suffers under gang scheduling and that may lead to significant CPU fragmentation. On other side a traditional Unix scheduler does good work in scheduling I/O bound threads since it gives high priority to I/O blocked threads when the data became available from disk. As those threads typically run for a small amount of time and then blocks again, giving them high priority means running the thread that will take the least amount of time before blocking, which is coherent to the theory of uniprocessors scheduling where the best scheduling strategy possible under total completion time is Shortest Job First [15]. In the local task scheduler of Concurrent Gang, such high priority is preserved. In practice the operation of the Concurrent Gang scheduler at each processor will proceed as follows: The reception of the global clock signal will generate an interruption that will make each processing element schedule threads as defined in the trace diagram. If a thread blocks, control will be passed to the one of the class 2 threads defined by the local task scheduler of the PE until the arrival of the next clock signal.

Differentiation between parallel jobs that may be gang scheduled and those that should not can be made by the user or through a heuristic algorithm. In Concurrent Gang we take the non-clairvoyant approach, where the scheduler itself has minimum information about the job - In our case processor count and memory requirements. In Concurrent Gang one possible algorithm for differentiation is the following heuristic: each local scheduler computes the average of slot utilization for each thread, that is, if a thread blocks due to I/O and it have used 20% of the time of its allocated slot, the slot utilization for that thread on that cycle was 0.20. If slot utilization falls below 0.50 due to I/O blocking for a 5 cycle average, then that thread is scheduled as a local thread. Observe that slot utilization is computed for even those parallel threads that are not gang scheduled at the moment - in that case the slot duration will correspond to the time quanta of the local scheduler. The slot allocated to the corresponding parallel thread will then be used by the local task scheduler until the thread is scheduled again as a gang.

We have chosen in Concurrent Gang to give priority to computing intensive applications and to try to to the best service possible for I/O bound applications. Then I/O bound applications will always be scheduled whether there is a slot under control of the local task scheduler and the required data from disk is available. In function of system needs, local task schedulers may have one or more slices dedicated to them, and those slices do not need to be consecutive in the trace diagram.

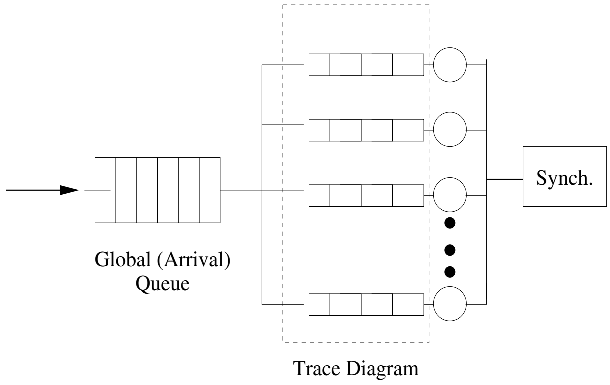

Figure 2. Modeling Concurrent Gang class algorithm

The revised version of the concurrent gang algorithm becomes then:

1 - Update Eligible thread list
2 - Allocate Threads of First Job in General Queue.
3 - While not end of Job Queue
    Allocate all threads of remaining parallel jobs using a defined spatial sharing strategy
4 - Run

Between Workload Changes

- Calculate the average slot utilization of each thread, including parallel threads scheduled as local threads
- If the average of a parallel thread fells below 0.5 schedule it as a local thread
- If the average of a parallel thread scheduled as a local thread gets greater than 0.5, reschedule the thread as a gang in its previous slot position.

3 Scalability and Space Sharing in Concurrent Gang

Concurrent Gang is a scalable algorithm due to the presence of a synchronizer working as a global clock, which allows the scheduler to be distributed among all processors. The front end is only activated in the event of a workload change, and decision in the front end are made as a function of the chosen space sharing strategy. As decisions about context switch are made locally, without relying on a centralized controller, concurrent gang schedulers with global clocks provide gang service in a scalable manner. This differs from typical gang scheduling implementation where job-wide context switch relies in a centralized controller, which limits scalability and efficient utilization of processors when a thread blocks. Another algorithm using gang service aimed at providing scalability is the Dynamic Hierarchical Control[7, 9]. However authors give no solution for the thread blocking problem. In Concurrent Gang, the distribution of the scheduler among all processors without any hierarchy allows each PE decide for itself to do if a thread blocks, without depending on any other PE.

To state and analyze space sharing startegies in Concurrent Gang, we prove a theorem that states that periodic schedules achieve better (or at least as good as) spatial utilization than non-periodic ones. That stated we may consider only finite trace diagrams in the remainder.

Theorem 1 *Given a workload W, for every temporal schedule S there exists a periodic schedule S_p such that the idling ratio of S_p is at most that of S,*

Proof - First of all, let's give a definition that will be useful in this proof. We define here *job happiness* in a interval of time as the number of slots allocated to a job divided by the total number of slots in the interval.

Define the progress of a job at a particular time as the number of slices granted to each of its threads up to that time. Thus, if a job has V threads, its progress at slice t may be represented by a progress vector of V components, where each component is an integer less than or equal to t. By the rules of legal execution, no thread may lag behind another thread of the same job by more than a constant C number of slices. Therefore, no two elements in the progress vector can differ by more than C. Define the differential progress of a job at a particular time as the number of slices by which each thread leads the slowest thread of the job. Thus a differential progress vector at time t is also a vector of V components, where each component is an integer less than or equal to C. The differential progress vector is obtained by subtracting out the minimum component of the progress vector from each component of the progress vector. The *system's differential progress vector* (SDPV) at time t is the concatenation of all job's differential progress vectors at time t. The key is to note that the SDPV can only assume a finite number of values. Therefore there exists an infinite sequence of times $t_{i_1}, t_{i_2}, ...$ such that the SDPVs at these times are identical.

Consider any time interval $[t_{i_k}, t_{i'_k}]$. One may construct a periodic schedule by cutting out the portion of the trace diagram between t_{i_k} e $t_{i'_k}$ and replicating it indefinitely along the time axis.

First of all, we claim that such a periodic schedule is legal. From the equality of the SPDVs at t_{i_k} e $t_{i'_k}$ it follows that all threads belonging to the same job receive the same number of slices during each period. In other words, at the end of each period, all the threads belonging to the same job have made equal progress. Therefore, no two threads lag behind another thread of the same job by more than a constant number of slices.

Secondly, observe that it is possible to choose a time interval $[t_{i_k}, t_{i'_k}]$ such that the happiness of each job in the during this interval is at least as much as in the complete trace diagram. This implies that the happiness of each job in the constructed periodic schedule is greater than or equal to the happiness of each job in the original temporal schedule.

Therefore, the idling ratio of the constructed periodic schedule must be less than or equal to the idling ration of the original temporal schedule. Since the fraction of area in the trace diagram covered by each job increases, the fraction covered by the idle slots must necessarily decrease. This concludes the proof.

A consequence of the previous theorem is stated in the following corollary:

Corollary 1 *Given a Workload W, for the set of all feasible periodic schedules S, the schedule with smaller idling ratio is the one with smaller period.*

Proof - The feasible schedule with smaller period is the one with has the smaller number of slices (resulting in a smaller number of total slots) which packs all jobs as defined in the Concurrent Gang algorithm. The number of occupied slots is the same for all feasible periodic schedules, since the workload is the same. So the ratio between the number of idle slots, which is the difference between the total number of slots and the number of occupied slots, and the total number of slots is minimized when we have a minimum number of total slots, which is the case in the minimum period schedule.

Packing in Concurrent Gang: In the classical, one dimensional bin-packing problem, a given list of items $L = I_1, I_2, I_3, ...$ is to be partitioned into a minimum number of subsets such that the items on each subset sum to no more than C. In the standard physical interpretation, we view the items of a subset as having been packed in a bin of capacity C. This problem is NP-Hard[1], so research has concentrated on algorithms that are merely close to optimal. For a given list L, let OPT(L) be the number of bins used in optimal packing, and define:

$$s(L) = \lceil \frac{\sum L}{C} \rceil \qquad (1)$$

Note for all lists L, $s(L) \leq OPT(L)$. For a given algorithm A, let A(L) denote the number of bins used when L is packed by A and define the waste $w_A(L)$ to be A(L)-S(L).

In this paper we deal with bin-packing algorithms that are dynamic (sometimes also dubbed "on-line"). A bin packing algorithm is dynamic if it assigns items to bins in order $(I_1, I_2, ...)$, with item I_i assigned solely on the basis of the sides of the preceding items and the bins which they are assigned, without reference to the size or number of remaining items.

Dynamic Competitive Ratio: In order to compare the various space sharing strategies we will use in this paper *competitive ratio*(CR) based metrics[2, 15]. The reason is because the competitive ratio is a formal way of evaluating algorithms that are limited in some way (e.g., limited information, computational power, number of preemptions)[2] what is indeed our case. This measure was first introduced in the study of a system memory management problem[14, 17]. The Competitive ratio[12, 2] for a scheduling algorithm A is defined as:

$$CR(n) = \sup_{J:|J|=n} \frac{A(J)}{OPT(J)} \quad (2)$$

Where $A(J)$ denotes the cost of the schedule produced by the algorithm A, and OPT(J) denotes the cost of the optimal scheduler, all under a predefined metric M. One way to interpret the competitive ratio is as the payoff to a game played between an algorithm A and an all-powerful malevolent adversary OPT that specifies the input J[12].

It is worth noting that the competitive ratio in previous cases were defined for a static scheduling. For the dynamic case, the definition of equation 2 is not convenient, since for a dynamic scheduling n can be of order of thousands or tens of thousands of jobs, but they are *spaced in time*, in a way that, at each instant of time, we would have typically tens of jobs at most scheduled in the machine. Besides that, competitive analysis has been criticized because it often yields ratios that are unrealistic high for "normal" inputs, since it consider the worst case workload, and as a result it can fail to identify the class of online algorithms that work well [12].

In this text we will be interested in the dynamic case, which is the case where we have different release times for different jobs. For the application of CR metric in dynamic scheduling, let's consider as reference (adversary) algorithm the optimal algorithm OPT for a predefined metric M applied at each new arrival epoch. The OPT scheduler will be a clairvoyant adversary, with unbounded number of preemptions and unbounded computational power. We will call this metric as Dynamic Competitive Ratio (CR_d), and the scheduler defined by applying OPT at arrival epochs as OPT_d. Formally we have:

$$CR_d(N) = \frac{1}{N} \sum_{\tau=1}^{N} \frac{A(\tau)}{OPT_d(\tau)} \quad (3)$$

Where N represents the number of arrivals epochs considered. Observe that CR_d only varies at arrival epochs. As Workload we have a (possibly infinite) sequence of jobs $J = \{J_1, J_2, J_3, J_4, ...\}$, with an arrival epochs $a_i \geq 0$ associated with each job. At the time of each arrival, workload changes are taken into consideration.

A very important question is the determination of the workload to be used in conjunction with CR_d to compare different algorithms. When choosing the new coming job at each arrival epoch, we have to possibilities : either selecting an "worst case" job that would maximize the CR_d or considering synthetic workload models with arrival epochs and running times being modeled as random variables. Since with the "worst case" option we may create fake workloads that never happen in practice and leads to the sort of criticism we cited before, we think that the best option is to use one of the synthetic workload models that have been recently proposed in the literature[3, 5]. These models and its parameters have been abstracted through careful analysis of real workload data from production machines. The objective with this approach is to produce an average case analysis of algorithms based on real distributions.

A lower bound for the dynamic competitive ratio is derived in the following theorem.

Theorem 2 *For the class C of non-clairvoyant schedulers,* $CR_d(N) \geq 1$ *,* $A \in C$*,* $N > 0$

Proof - Consider N=1. In the case of a workload composed by one embarrassingly parallel job with a degree of parallelism P, if we have a non clairvoyant scheduler that schedules each thread in a different processor, as would do OPT_d, we have $CR_d(1) = 1$.

Conversely, an optimal clairvoyant scheduler will always be capable of producing a scheduling at least as good as a non-clairvoyant one, since the clairvoyant scheduler has all the information available about the workload at any instant in time. So $CR_d(N) \geq 1$

For bin-packing, the reference or optimal algorithms will be simply the sum of item sizes s(L), since $s(L) \leq OPT(L)$, and it can be easily computed. However, in order to used dynamic CR to compare the performance of algorithms, we must first define precisely the workload model we will use in the dynamic CR computation, which is done in the next section.

Workload Model: The workload model that we consider in this paper was proposed in [3]. This is a statistical model of the workload observed on a 322-node partition of the CTC SP2 from June 25, 1996 to September 12, 1996, and it is intended to model rigid job behavior. During this period, 17440 jobs were executed.

The model is based on finding Hyper-Erlang distributions of common order that match the first three moments of the observed distributions. Such distributions are characterized by 4 parameters:

- **p** – the probability of selecting the first branch of the distribution. The second branch is selected with probability 1 - p.

- λ_1 – the constant in the exponential distribution that forms each stage of the first branch.

- λ_2 – the constant in the exponential distribution that forms each stage of the second branch.

- **n** – the number of stages, which is the same in both branches.

As the characteristics of jobs with different degrees of parallelism differ, the full range of degrees of parallelism is first divided into subranges. This is done based on powers of two. A separate model of the inter arrival times and the service times (runtimes) is found for each range. The defined ranges are 1, 2, 3-4, 5-8, 9-16, 17-32, 33-64, 65-128, 129-256 and 257-322.

Tables with all the parameter values are available in [3].

Dynamic CR applied to First Fit and Best Fit: We conducted experiments, using dynamic CR, with first fit and best fit strategies. They consisted of computing the dynamic CR for a sequence of SPMD jobs generated through the model described in the previous section. The machine considered has 128 processing elements, and the size of jobs varied between 1 and 128, divided in 8 ranges. Then the first and best fit strategies were applied, and the number of slices used by each algorithm for each job arrival was computed. This number was then divided by the number of slices that would be used considering the sum s(L) as defined in equation 1. The smaller the number of slices, the smaller the period is, with more slots dedicated to each job over time.Two cases were simulated:

- With job migration - All jobs return to the central queue and are redistributed among all processors at each workload change.

- Without job migration - In this case an arriving job is allocated accordingly to a given algorithm without changing the placement of other jobs.

Two sequences of jobs were randomly generated: One with 7000 jobs and other with 40000. Results indicate that maximum spread between first fit and best fit is 3% and first fit can be used with no system degradation.

References

[1] E.G. Coffman, D.S. Johnson, P.W. Shor, and R.R. Weber. Markov Chains, Computer Proofs, and Average Case Analysis of Best Fit Bin Packing. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 412–421, 1993.

[2] J. Edmonds, D.D. Chinn, T. Brecht, and X. Deng. Non-Clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics (extended abstract). In *Proceedings of the 1997 ACM Symposium of Theory of Computing*, pages 120–129, 1997.

[3] J. Jann et al. Modeling of Workloads in MPP. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:95–116, 1997.

[4] Patrick G. Solbalvarro et al. Dynamic Coscheduling on Workstation Clusters. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:231–256, 1998.

[5] D. Feitelson. Packing Schemes for Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162:89–110, 1996.

[6] D. Feitelson and M. A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:238–261, 1997.

[7] D. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, pages 65–77, May 1990.

[8] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[9] D. Feitelson and L. Rudolph. Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control. *Journal of Parallel and Distributed Computing*, 35:18–34, 1996.

[10] D. Feitelson and L. Rudolph. Metrics and Bechmarking for Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:1–24, 1998.

[11] M. A. Jette. Performance Characteristics of Gang Scheduling In Multiprogrammed Environments. In *Proceedings of SC'97*, 1997.

[12] B. Kalyanasundaram and K. Pruhs. Speed is as Powerful as Clairvoyance. In *Proceedings of the 36th Symposium on Computer Science*, pages 214–221, 1995.

[13] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:215–237, 1997.

[14] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On Line problems. In *Proceedings of the Twentieth Annual Symposium on the theory of Computing*, pages 322–333, 1988.

[15] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[16] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Comp. Systems*, pages 22–30, 1982.

[17] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202 – 208, 1985.

[18] L. G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8):103 – 111, 1990.