Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Towards Flexibility and Scalability in Parallel Job Scheduling

Fabricio Alves Barbosa da Silva * Isaac D. Scherson†

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

Laboratoire ASIM, LIP6
Universite Pierre et Marie Curie
Paris, France

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 tasks of a job are scheduled for simultaneous execution on a partition of a parallel computer. Gang Scheduling has many advantages, such as responsiveness, efficient sharing of resources and ease of programming. However, there are two major problems associated with gang scheduling: scalability and the decision of what to do when a task blocks. 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 with different characteristics. Besides that, scalability in Concurrent Gang is achieved through the use of a global clock that coordinates the gang scheduler among different processors.

Key Words: Parallel job scheduling, gang scheduling, parallel computation

1 Introduction

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern parallel computers. It is defined as: "Given the aggregate of all tasks of multiple jobs in a parallel system, find a spatial and temporal allocation to execute all tasks efficiently". Each job in a parallel machine is composed by one or more tasks. 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 [7]. 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 tasks 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 [3], efficient sharing of resources[8] and ease of programming. In gang service the tasks of a job are supplied with an environment that is very similar to a dedicated machine [8]. 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[5]. 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[2]. 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 similar to the one defined in the BSP model [14]. 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.

This paper is organized as follows: the Concurrent Gang algorithm is described in section II. Scalability issues in Concurrent gang are discussed in section III. Experimental results are in section IV and section V contain our final remarks.

[1] Supported by Capes, Brazilian Government, grant number 1897/95-11.
[2] Supported in part by the Irvine Research Unit in Advanced Computing and NASA under grant #NAG5-3692.

2 Concurrent Gang

In this section we describe the Concurrent Gang algorithm. First we describe the time utilization under Concurrent Gang. Then, the task classification that is made by the algorithm is described; we shall see that this task classification is used by Concurrent Gang to decide locally which task to schedule if the current task blocks. Then the algorithm itself is detailed, with the description of the components of a Concurrent Gang Scheduler.

2.1 Time Utilization

In parallel job scheduling, as the number of processors is greater 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 [11]. 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 tasks 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 is 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. Observe that the duration of the slice for Concurrent Gang is defined by the period of the global clock. 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 task 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 that 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.

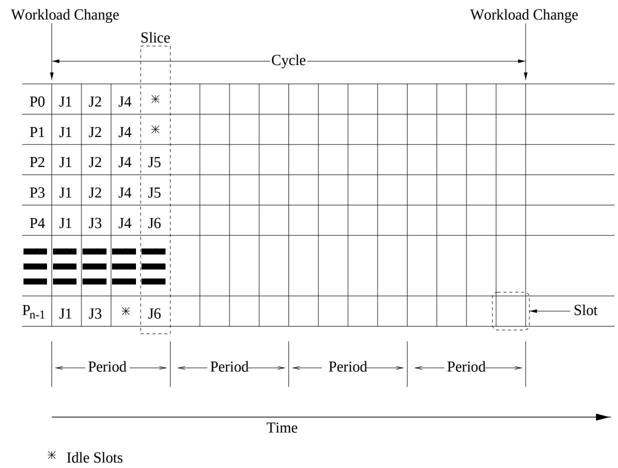

Figure 1: Definition of slice, slot, period and cycle

2.2 Task Classification

In Concurrent Gang, each PE classifies each one of its allocated tasks into classes. Examples of such classes are: I/O intensive, Synchronization intensive, and computation intensive. Each one of these classes is similar to a fuzzy set [15]. A fuzzy set associated with a class A is characterized by a membership function $f_A(x)$ with associates each task T to a real number in the interval [0,1], with the value of $f_A(T)$ representing the "grade of membership" of T in A. Thus, the nearer the value of $F_A(T)$ to unity, the higher the grade of membership of T in A. For instance, consider the class of I/O intensive tasks, with its respective characteristic function $f_{IO}(T)$. A value of $f_{IO}(T) = 1$ indicates that the task T only have I/O statements, while a value of $f_{IO}(x) = 0$ indicates that the task T have executed no I/O statement at all.

The value of $f(T)$ for a class is computed by the PE at the end of the slot dedicated to a task. As an example, let's consider the I/O intensive class. The computation is made as follows: At the reception of the global clock signal, the scheduler computes the machine utilization of each task that was scheduled in the previous global clock period. For instance if at the beginning of the period one task was scheduled and after 20% of the duration of the period the task blocks due to an I/O command, the computation time of that task was .20 and the I/O time was .80. A average of the time spent in I/O is then made over the last five times where the task was scheduled. This average determines the grade of membership of a particular task to the class I/O intensive. As many jobs proceed in phases, the reason for using an average over the last five times a task was scheduled is detection of phase change. If a task changes from a I/O intensive phase to a computation intensive phase, this change should be detected by the Concurrent Gang scheduler.

In principle, four major classes are possible: I/O intensive, Computing intensive, Communications (point to point) intensive and synchronization intensive. We will see in the next subsection that only a subset of them are used in Concurrent Gang.

2.3 Definition of Concurrent Gang

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 tasks 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 tasks in a processor as defined in the trace diagram is made through local schedulers, independently of the front end.

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 task indicated in the trace diagram at the arrival of a synchronization signal. The local task scheduler is responsible for scheduling specific tasks (as described in the next paragraph) allocated to a PE 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 types of tasks in a concurrent gang scheduler: Those that should be scheduled as a gang with other tasks in other processors and those that gang scheduling is not mandatory. Examples of the first class are tasks that compose a job with fine grain synchronization interactions [5] and communication intensive jobs[2]. Second class task examples are local tasks or tasks that compose an I/O bound parallel job, for instance. In [9] 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 tasks since it gives high priority to I/O blocked tasks when the data became available from disk. As those tasks typically run for a small amount of time and then blocks again, giving them high priority means running the task 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 [10]. In the local task scheduler of Concurrent Gang, such high priority is preserved. Another example of jobs where gang scheduling is not mandatory are embarrassingly parallel jobs. As the number of iterations among tasks belonging to this class of jobs are small, the basic requirement for scheduling a embarrassingly parallel job is to give those jobs the greater fraction of CPU time possible, even in an uncoordinated manner.

Differentiation among tasks that should be gang scheduled and those that a more flexible scheduler is better can be made by the user or using the grade of membership information computed by the local scheduler (as explained in the last subsection) about each task associated with the respective processor. In Concurrent Gang we take the non-clairvoyant approach (i.e. the grade of membership approach), where the scheduler itself has minimum information about the job - In our case processor count and memory requirements.

The local task scheduler defines a priority for each task allocated to the corresponding PE. The priority of each task is defined based on the grade of membership of a task to each one of the major classes described in the previous subsection. Formally, the priority of a task T in a PE is defined as:

$$Pr(T) = max(\alpha \times \lambda_{IO}, \lambda_{COMP}) \quad (1)$$

Where $\lambda_{IO}, \lambda_{COMP}$ are the grade for membership of task T to the classes I/O intensive and Computation intensive. The objective of the parameter α is to give greater priority to I/O bound jobs ($\alpha > 1$). In our experiments in this work we have defined $\alpha = 2$. The choices made in equation 1 intend to give high priority to I/O intensive jobs and computation intensive job, since such jobs can benefit the most from uncoordinated scheduling. The multiplication factor α for the class I/O intensive gives higher priority to I/O bound tasks over computation intensive tasks, since those jobs have a greater probably to block when scheduled than computing bound tasks. By other side, synchronization intensive and communication intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization[5, 2]. A synchronization intensive or communication intensive phase will reflect negatively over the grade of membership of the class computation intensive, reducing the possibility of a task be scheduled by the local task scheduler. Among a set of tasks of the same priority, the local task scheduler uses a round robin strategy.

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 tasks as defined in the trace diagram. If a task blocks, control will be passed to the one of the other tasks allocated in the PE defined by the local task scheduler of the PE in function of the priority assigned to each one of the tasks until the arrival of the next clock signal. The task chosen is the one with greater priority.

In the event of a job arrival, a job termination or a job changing its number of eligible tasks (events which define effectively a workload change if we consider moldable jobs) the front end Concurrent Gang Scheduler will :

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

Between Workload Changes

- If a task blocks or in the case of an idle slot, the local task scheduler is activated, and it will decide to schedule a new task based on:

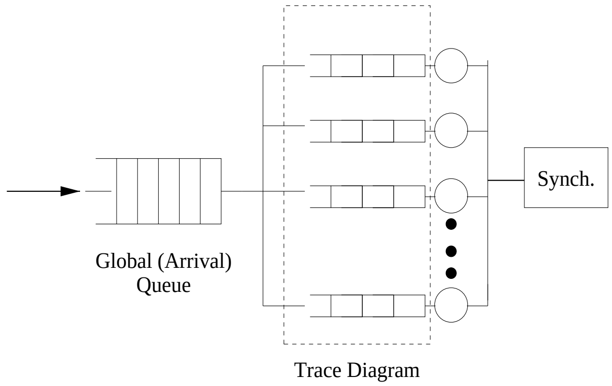

Figure 2: Modeling Concurrent Gang class algorithm

- Availability of the task (task ready)
- Priority of the task defined by the local task scheduler.

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.

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 tasks 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, slices are scanned in serial order until a slice with sufficient capacity is found. In best fit, the slices are sorted according to their capacities. The one with the smallest sufficient capacity then is chosen.

In the case of creation of a new task by a parallel task, or parallel task completion, 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.

3 Scalability 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 is 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 task blocks. Another algorithm using gang service aimed at providing scalability is the Distributed Hierarchical Control[4, 6]. However authors give no solution for the task 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 task blocks, without depending on any other PE.

4 Experimental Results

The performance of Concurrent Gang was simulated and compared with the traditional gang scheduling algorithm, using first fit without thread migration space sharing strategy in both cases. The reason of using first fit is that it was proven in [13] that this strategy can be used with no system degradation if compared with other bin-packing policies (without thread migration) given the workload model defined in [1], besides its smaller complexity. First we describe the simulator, then we detail the workload model used, and finally simulation results are presented and analyzed.

4.1 Description of the Simulator

To perform the actual experiments we used a general purpose event driven simulator, first described in [12], being developed by our research group for studying a variety of problems (e.g., dynamic scheduling, load balancing, etc). The format used for describing jobs (composed by a set of task) is a set of parameters used to describe the job characteristics such as computation/communication ratio through probabilistic distributions. The actual communication type, timing and pattern (with whom a particular task from a job will communicate with) are then left unspecified and it is up to the simulator to convert this user specification into a DAG, using the probabilistic distributions provided by the user, for each of the parameters. Other parameters include the spawning factor for each task, a task life span, synchronization pattern, degree of parallelism (maximum number of task that can be executed at any given time), depth of critical path, etc. Please notice that even though probabilistic distributions are used to generate the DAG, the DAG itself behaves in a completely deterministic way.

Once the input is in the form of a DAG, and the module responsible for implementing a particular scheduling algorithm is plugged into the simulator, several experiences can be performed using the same input by changing some of the parameters of the simulation such as the number of processing elements available, the topology of the network, among others, and their outputs, in a variety of formats, are recorded in a file for later visualization. The simulator offers a gamut of features aimed at simplifying the task of the algorithm developer. For the case of dynamic scheduling the simulator offers among others methods for manipulating partitions (creation, deletion, and resizing), entire job manipulation (suspension, execution), as well as task level selection, message storing and forwarding, deadlock free communication and synchronization, etc.

4.2 Workload Model

The workload model that we consider in this paper was proposed in [1]. This is a statistical model of the workload observed on a 322-node partition of the Cornell Theory Center's IBM 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 [1].

4.3 Simulation Results

We simulated a 32-processor machine in a mesh configuration. Six of the job size ranges described the previous section were used. Five different workloads were considered. The first workload is composed exclusively by I/O intensive jobs. We also simulated workloads composed exclusively by computation intensive, communication intensive and synchronization intensive jobs. The fifth workload is composed by a mix of synchronization intensive, computing intensive, I/O bound and communication intensive jobs, with inter-arrival and execution times of jobs given by Hyper Erlang Distributions. All workloads are randomly generated, and then the same set of jobs with their arrival and execution times are presented to both a Concurrent gang scheduler and a Gang Scheduler. Space sharing strategy in all cases is first fit without thread migration. At the end of each simulation, the total idle time and number of completed jobs are returned. It should be noted that the total idle time in the simulations is not composed be idle slots only, but also by the time which a particular task was waiting for I/O, synchronization and communication completion. Context switch time was not considered on simulations. The reason is that context switch overhead is highly dependent on operating system implementation and processor architecture.

Simulation results for the I/O intensive workload are shown in tables 1 and 2. We an observe a significant improvement over gang scheduling, both in throughput (jobs completed by unit of time) and total idle time. This results was expected, since Concurrent Gang provides grater flexibility than gang scheduling, which is necessary for this kind of job.

Table 1: Experimental results - I/O intensive workload

Simulation time	Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	2	72
10000	6	68
20000	14	67
30000	23	66
40000	28	66

Table 2: Experimental results - I/O intensive workload

Simulation time	Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	5	43
10000	10	26
20000	19	14
30000	34	10
40000	45	7

Tables 3 and 4 show results of the simulation of a Computation intensive workload, where the jobs have no communication, I/O or synchronization statements at all. Also in this case Concurrent Gang performs better tat Gang scheduling. The reason is that Concurrent Gang uses the idle slots of gang scheduling to schedule computing intensive tasks that the local task scheduler on each PE detects at run-time.

Table 3: Experimental results - Computation intensive workload

Simulation time	Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	5	24
10000	11	17
20000	23	12
30000	35	11
40000	44	9

In tables 5 and 6, results for communication intensive (point to point) workload are shown. Once again we observe a significant improvement in Total Idle time and in throughput. Although gang scheduling is better than uncoordinated scheduling for this kind of job, Concurrent Gang

Table 4: Experimental results - Computation intensive workload

Simulation time	Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	8	10
10000	12	6
20000	25	4
30000	37	3
40000	48	2

shown to be even better that gang scheduling. The communication semantics used was asynchronous non blocking for send statements and blocking for receive statements. The better performance of concurrent gang is due to the fact that it will try to schedule each ready task as soon as possible, in function of its priority. This minimizes the receiver's blocking time if compared with gang scheduling, where each task will only be scheduled a fixed number of times (generally 1) on each cycle.

Table 5: Experimental results - Communication intensive workload

Simulation time	Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	3	43
10000	6	39
20000	17	34
30000	31	32
40000	35	31

Table 6: Experimental results - Communication intensive workload

Simulation time	Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	5	27
10000	15	14
20000	33	8
30000	48	5
40000	63	4

Synchronization intensive workload simulation results are shown in tables 7 and 8. The synchronization was always global, i.e. over all tasks of a job. Again, although gang scheduler is a better option to scheduled those jobs than uncoordinated scheduled, concurrent gang is shown to be even better that gang scheduling, having significant improvements both in total idle time and throughput. This is due to the fact that rescheduling those tasks that have not reached the barrier on its originally assigned slot as soon as possible allows those tasks to reach the barrier faster than if they would be scheduled only on the next cycle. As a consequence, the job can pass by the barrier earlier than with standard gang.

Table 7: Experimental results - Synchronization intensive workload

Simulation time	Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	4	46
10000	14	43
20000	24	40
30000	41	39
40000	47	39

Table 8: Experimental results - Synchronization intensive workload

Simulation time	Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)
5000	3	10
10000	16	6
20000	31	4
30000	54	3
40000	75	2

The last workload simulated was composed by a mix of jobs with different characteristics varying from I/O intensive to synchronization intensive. with some jobs also composed by a mix of different kind of statements. The results are shown on tables 9 and 10. The improvement of gang scheduling is significant, both in machine utilization and in throughput.

Table 9: Experimental results - Mixed Workload - Gang Scheduling

Simulation time	Gang	
Seconds	Jobs Completed	Total Idle Time (%)
10000	4	68
20000	8	67
30000	9	67
40000	9	67
50000	10	67

It is clear by the figures in tables 9 and 10 that Concurrent Gang outperforms the traditional gang scheduling algorithm both in utilization and throughput for the mixed workload. Again, this is due to the action of the local scheduler on each PE, that tries to schedule a eligible task every time the current task blocks using the criteria previously defined.

5 Discussion and Conclusion

In this paper we presented a new parallel scheduling algorithm dubbed Concurrent Gang. The main differences over standard gang scheduling are the explicit definition of a external global clock, which can be either hardware and software implemented, and the presence of local task scheduler

Table 10: Experimental results - Mixed Workload - Concurrent Gang

Simulation time	Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)
10000	4	33
20000	10	23
30000	19	18
40000	24	15
50000	35	13

which decides what to do if a task of the job scheduled as a gang blocks.

The concurrent gang approach are more beneficial to workloads that require a more flexible scheduling than is possible with gang scheduling. An example is I/O bound workloads, as is demonstrated with simulation results. For workloads requiring coordinated scheduled, the Concurrent Gang algorithm becomes equivalent to the standard gang scheduler, as verified with a communication bound workload.

The idle time in Concurrent Gang is reduced because, in the event of an idle slot, Concurrent Gang always tries to schedule other tasks that are either local tasks (although local tasks are not considered in the simulations) or tasks that do not require, at that moment, coordinated scheduling with other tasks of the same job. This is the case, for instance, of I/O intensive tasks and Computation intensive tasks. Observe that idle time can be further reduced through the use of a space sharing strategy with thread migration.

We considered in this paper a parallel machine working as a general purpose, multiuser, multiprogrammed server, The workload considered in the simulations could be considered as a non-memory demanding workload: We suppose that each PE has sufficient memory to accommodate all tasks allocated for that processor at a time, or a efficient virtual memory system minimizes the effects of insufficient memory. Further work will consider the use of Concurrent Gang with heavy workloads, where all tasks have large memory requirements.

References

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

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

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

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

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

[6] 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.

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

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

[9] 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.

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

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

[12] F.A.B. Silva, L.M. Campos, and I.D. Scherson. A Lower Bound for Dynamic Scheduling of Data Parallel Programs. In *Proceedings EUROPAR'98*, 1998.

[13] F.A.B. Silva and I.D. Scherson. Improvements in Parallel Job Scheduling Using Gang Service. In *Proceedings 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, 1999.

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

[15] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.