Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang

Fabricio Alves Barbosa da Silva
Laboratoire ASIM, LIP6
Universite Pierre et Marie Curie
Paris, France
fabricio.silva@lip6.fr

Isaac D. Scherson
Information and Comp. Science
University of California
Irvine, CA 92697 U.S.A.
isaac@uci.edu

Abstract

In this paper we propose a new class of scheduling policies, dubbed Concurrent Gang, that combines the advantages of gang scheduling for communication and synchronization intensive parallel jobs with the flexibility of a Unix scheduler for sequential and I/O intensive jobs. Besides that, scalability in Concurrent Gang is achieved through the use of a global synchronizer that coordinates the gang scheduler among different processors. Simulation results are provided comparing the performance of Concurrent Gang with Gang Scheduling and show significant performance improvements, in particular for I/O bound jobs.

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. 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. 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 [1], efficient sharing of resources[2] 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 [2]. 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[3]. 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. In order to do that, the Concurrent Gang scheduler identifies the characteristics of each task at run time and takes a decision about the best way of scheduling the specific tasks in function of those characteristics. Along with that, a solution to the problem related to gang schedulers of what to do when a task blocks is proposed.

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 send a synchronization (clock) signal to all PEs at regular intervals of L time units. This architecture is similar to the one defined in the BSP model [5].

This paper is organized as follows: the Concurrent Gang algorithm is described in section 2. Experimental results are in section 3 and section 4 contains our final remarks.

1.1 Terminology

In parallel job scheduling, the time utilization as well as the spatial utilization can be better visualized with the help of a bidimensional diagram dubbed *trace diagram* (this diagram is also known in the literature as Ousterhout matrix [6]). One dimension represents processors while the

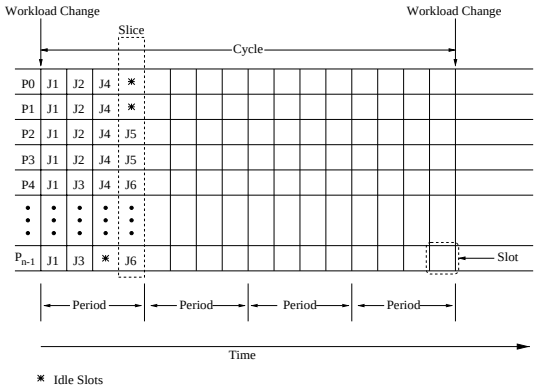

Figure 1. Definition of slice, slot, period and cycle. J1 stands for job 1, J2 for job 2, etc. Job 2 is composed by 4 tasks

other dimension represents time. 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 definitions 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. We define *cycle* as the time between workload changes. 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. 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*.

In this paper we will consider that a parallel *job* is composed by a set of parallel *tasks* that are execution entities derived from the same single SPMD program.

2 Concurrent Gang

In this section we describe the Concurrent Gang algorithm. We first describe the task classification that is made by the algorithm. Then the algorithm itself is detailed, with the description of the components of a Concurrent Gang Scheduler.

2.1 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 [7]. A fuzzy set associated with a class A is characterized by a membership function $f_A(x)$ which 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 $f_{IO}(T) = 0$ 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 time spent in I/O statements of each task that was scheduled in the previous slice. An average of the time spent in I/O is then made over the last five measures. 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 an I/O intensive phase to a computation intensive phase, this change should be detected by the Concurrent Gang scheduler.

2.2 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 corresponding 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 context 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 sequential tasks and parallel tasks that do not need global coordination, as described in the next paragraph, and it is similar to a UNIX scheduler. The Gang Scheduler has precedence over the local task scheduler.

We may consider two types of parallel tasks in a concurrent gang scheduler: Those that require coordinated scheduling 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 [3] and communication intensive jobs[4]. Second class task examples are local tasks or tasks that compose an I/O bound parallel job, for instance. In [8] 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 the 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 becomes 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 [9]. 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.

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 grades of membership of task T to the classes I/O intensive and Computation intensive respectively. The objective of the parameter α is to give greater priority to I/O bound jobs ($\alpha > 1$). In the experiments of this work we have defined $\alpha = 2$. The choices made in equation 1 intend to give high priority to I/O intensive and computation intensive jobs, 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 probability to block when scheduled than computing bound tasks. On the other hand, synchronization intensive and communication intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization[3, 4]. 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. The local task scheduler also defines a minimum priority β. If no parallel task has priority larger than β, the local task scheduler considers that all tasks are either communication or synchronization intensive, thus requiring coordinated scheduling.

In practice the operation of the Concurrent Gang scheduler in 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 a gang as defined in the trace diagram. If a task blocks, control will be passed to the local task scheduler that will schedule one of the other tasks allocated 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 larger priority, and the priority should be larger than β.

In the event of a job arrival, a job termination or a job changing its number of eligible tasks 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:

a - Availability of the task (task ready)

b - Priority of the task defined by the local task scheduler (priority $> \beta$).

The local queue positions represent 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.

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.

Scalability in Concurrent Gang is improved 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 or server, 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

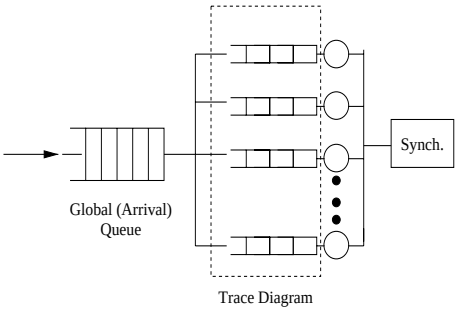

Figure 2. Modeling Concurrent Gang class algorithm

controller, which limits scalability and efficient utilization of processors when a task blocks.

3 Experimental Results

The performance of Concurrent Gang was simulated and compared with the traditional gang scheduling algorithm, using first fit without thread migration as space sharing strategy in both cases. First the simulation methodology is explained and then simulation results are presented and analyzed.

3.1 Simulation Methodology

To perform the actual experiments we used an improved version of a general purpose event-driven simulator, first described in [10], being developed by our research group for studying a variety of problems (e.g., dynamic scheduling, load balancing, etc).

The workload model that we consider in this paper was proposed in [11]. 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.

The model is based on finding Hyper-Erlang distributions of common order for inter-arrival and service times that match the first three moments of the observed distributions. 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.

Four classes of workloads were used in simulations. Our intention is to represent the major classes we considered in the description of Concurrent Gang. They are:

1 - Communication Intensive - In this workload all jobs are communication intensive, i.e. for each five statements four are local computation statements and one represents a point to point communication statement (send or receive). The semantics used for the point to point communication was non blocking asynchronous sends and blocking receives.

2 - Computation intensive - In this class all jobs only have local computation instructions.

3 - IO intensive - All jobs that belong to this workload execute one IO statement for ten local computation statements. One thread that has issued an IO statement remains blocked until IO completion.

4 - Synchronization intensive - In this workload there is one global synchronization statement for an average of four local computation statements. The synchronization is always global, that is, involves all tasks of a job.

We have simulated machines with 16 and 32 processors using as interconnection network a 2D torus. The simulator's time unit is seconds, and results are obtained for 10000, 20000, 30000 and 40000 seconds.

All workloads are randomly generated, and then the same set of jobs with their arrival and execution times are presented to Concurrent gang scheduler, a Concurrent Gang scheduler without the priority mechanism as defined in section 2 (in this case a round robin strategy for defining which job will run next is used) and a Gang Scheduler. Space sharing strategy for the Gang scheduler and the concurrent gang scheduler 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 by idle slots only, but also by the time which a particular task was waiting for I/O, synchronization and communication completion.

3.2 Simulation Results

In this section we present results of simulations for the machine with 16 processors. Comparison between the three algorithms for simulations of a 32 processors' machine are equivalent to those presented for a 16 processors' machine.

Simulation results for the I/O intensive workload are shown in figure 3 for 16 processors. We can observe a significant improvement over gang scheduling, both in throughput (jobs completed by unit of time) and total idle time, with Concurrent Gang with priorities having a better performance. These results were expected, since Concurrent Gang provides larger flexibility than gang scheduling, which is necessary for this kind of job. The Idle time associated with each algorithm is shown at figures 4.

Figure 5 shows results of the simulation of a Computation intensive workload, where the jobs have no communication, I/O or synchronization statements at all. Figure 5

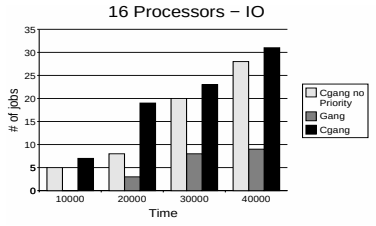

Figure 3. 16 Processors, I/O bound workload: Throughput

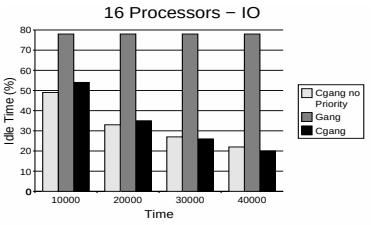

Figure 4. 16 Processors, I/O bound workload: idle time

shows the results for 16 processors. Also in this case Concurrent Gang performs slightly better than 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. The idle time for the 16 processors machine is shown in figure 6.

In figures 7, results for communication intensive (point to point) workload are shown for 16 processors. 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 shows to be even better than gang scheduling. 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. Idle times are in figures 8.

Synchronization intensive workload simulation results are shown in tables 9 for 16 processors. The synchronization was always global, i.e. over all tasks of a job. Again, although gang scheduler is a better option to schedule 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. Observe also that Concurrent Gang without priorities is worse than Concurrent Gang and Gang scheduling. The idle time for the 16 processors machine is shown in figure 10.

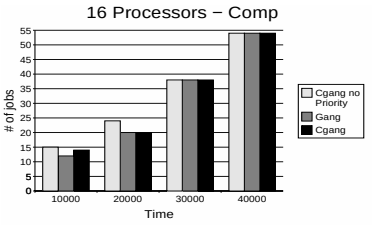

Figure 5. 16 Processors, Computation intensive workload: Throughput

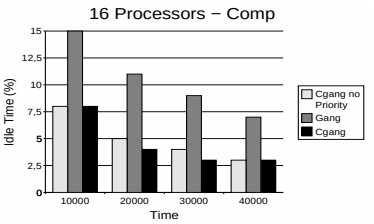

Figure 6. 16 Processors, Computation intensive workload: idle time

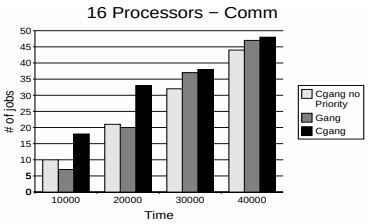

Figure 7. 16 Processors, Communication intensive workload: Throughput

4 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

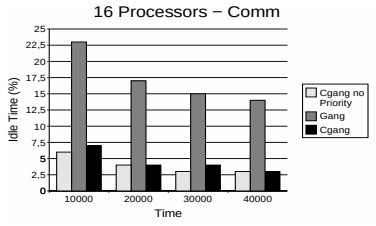

Figure 8. 16 Processors, Communication intensive workload: idle time

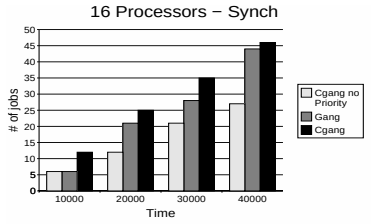

Figure 9. 16 Processors, Synchronization intensive workload: Throughput

an external global synchronizer and the presence of local task scheduler which decides what to do if a task of the job scheduled as a gang blocks, as we propose in this paper a solution to the task blocking problem in gang scheduling.

The Utilization in Concurrent Gang is improved because, in the event of an idle slot or blocked task, Concurrent Gang always tries to schedule 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. The priority mechanism in concurrent gang is used by the scheduler to make a smarter choice of the task to schedule in a idle slot, thus improving performance over a concurrent gang scheduler with no priority as shown in simulations.

5 Acknowledgments

The first author is supported by Capes, Brazilian Government, grant number 1897/95-11. The second author is supported in part by the Irvine Research Unit in Advanced Computing and NASA under grant #NAG5-3692. The authors would like to thank Luis Miguel Campos for useful discussions and his work in the development of the first version of the simulator.

References

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

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

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

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

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

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

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

[8] 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.

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

[10] 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.

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

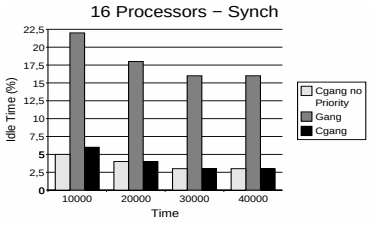

Figure 10. 16 Processors, Synchronization intensive workload: idle time