Improvements in Gang Scheduling for Parallel Supercomputers

Fabricio Alves Barbosa da Silva[1,*] *Luis Miguel Campos*[2]

Isaac D. Scherson[1,2,†]

fabricio.silva@asim.lip6.fr, {lcampos,isaac}@ics.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. The resulting policy is simulated and compared with gang scheduling using a general purpose event driven simulator specially developed for this purpose.

1 Introduction

Gang Scheduling [1][5] has been proposed as a practical solution to the dynamic parallel job scheduling problem. Dynamic means that the possibility of arbitrary arrival times for new jobs is allowed. A parallel job scheduler in general is responsible for finding a good scheduling allocation, both temporal and spatial, as a function of the existing workload. The temporal and spatial allocation represent the two dimensions in which computing resources are shared : the temporal sharing is also known as time slicing or preemption; and the space sharing is also known as space slicing and partitioning. These two classifications are orthogonal, and may lead to a taxonomy based on all possible combinations [1].

In gang scheduling each thread of execution of a parallel job is scheduled on an independent processor. The threads of a job are supplied with an environment that is very similar to a dedicated machine [4][5], and may or may not use all available processors.

In this paper we propose a class of scheduling policies, dubbed Concurrent Gang. It 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 [2], efficient sharing of resources, ease of programming, fine grain synchronization performance benefits [4], etc. are maintained.

This paper is organized as follows : In section 2 the general class of Concurrent Gang policies is described. In section 3 space sharing under Concurrent Gang is considered, with the definition of important concepts for the precise description of the space sharing strategy used. Section 4 gives the simulation results of Concurrent Gang with first fit as space sharing strategy, with the

[*]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.

respective analysis and comparison with gang scheduling.

2 Concurrent Gang

In gang scheduling, each job is allocated to the whole parallel machine for a time slice before being preempted. However, not all jobs necessarily use all of the machine's processors at all times. In order to further optimize the use of massively parallel systems the operating system must support scheduling policies aimed at scheduling simultaneously several jobs of different sizes and with no predefined arrival times.

Let us consider a machine with N processors in a MIMD architectural model as described in [6]. In Concurrent Gang the machine is shared by more than one job at any given time. All the jobs running concurrently are preempted by the end of time slice. The scheduler is responsible for providing an efficient machine utilization, both in temporal and spatial dimensions, always gang scheduling each job on available resources.

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 queue of eligible threads to execute. The mapping and allocation of threads of a job from the general queue to the processors' queues effected by the scheduler. 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) the Concurrent Gang Scheduler will :

1 Update Eligible thread list

2 Allocate Threads of First Job of General Queue in the required number of processors.

3 While not end of Job Queue

- Allocate all threads of remaining jobs using a defined spatial sharing strategy

4 Run

It should be noted that this algorithm leads to a bidimensional diagram, where one dimension corresponds to the number of processors, and another dimension is time. As each job is gang scheduled, it allocates the necessary number of processors in a given slot of time. As we suppose that the number of jobs in any given moment is finite, the time dimension is also finite, with a diagram defining a period that repeats itself if there is no change in the number and/or corresponding requirements of the jobs.

A variation of Concurrent Gang was proposed in [7], where jobs are preempted after all threads, running in parallel, execute a fixed number of instructions. The reason behind the preemption of the jobs after executing a fixed number of instructions is because it provides better performance under the competitive ratio metrics [7, 6]. In that case, a time slice can vary as a function of the characteristics of the job.

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.

Concurrent Gang implies a better performance and machine utilization than pure gang scheduling, since gang scheduled jobs may not use all processors, resulting in a smaller rate of processor utilization. Hence, Concurrent Gang space sharing is proposed to improve the utilization of individual processors in a parallel machine by combining the best characteristics of gang scheduling and partitioning and assuming current MIMD machines. Besides that, the execution of jobs in parallel implies also better overall execution times, since we are not obliged to always create a new time slice for a new coming job, but running that job in parallel with another already-running job, provided that there is a sufficient number of processors in the time slice. The queueing system's approach of Concurrent Gang provides a general framework for describing space sharing strategies based on gang scheduling, considering or not thread migration, given the capability of queueing systems to model the workload-resources interaction.

However, the description of a scheduler under Concurrent Gang is not complete if a space sharing strategy is not defined. In the next section we state some important concepts that are use-

ful for this definition and give some examples of Concurrent Gang schedulers.

3 Space sharing in Concurrent Gang

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, best fit and greedy policies. First fit and best fit policies were originally defined by Feitelson [3].

To clarify the application of these policies in Concurrent Gang let us first state some important concepts. These are the concepts of cycle, slice, period and slot. Figure 2 illustrates these definitions. 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, Concurrent Gang scheduling is periodic, with a period that is a function of the workload and the spatial allocation. A period is composed of slices; a slice corresponds to a time slice as in gang scheduling, with the difference that in Concurrent Gang we may have more than one job simultaneously scheduled in a slice. 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 bidimensional diagram showed in figure 3 is inherent to the concurrent gang algorithm, and it is used to define the spatial allocation strategy. We refer to this diagram as the *trace diagram*.

The implementation of Concurrent Gang with *first fit with thread migration* is a first example of a Concurrent Gang scheduler. It is based on a greedy algorithm applied at the time of a workload change. During the cycle, the workload is obviously assumed constant. Thus, the eligible threads of queued jobs are allocated to processors using the first fit strategy for each slice. Clearly, after all eligible threads are scheduled on a processor for some slice (slot), the temporal sequence is repeated periodically until a workload

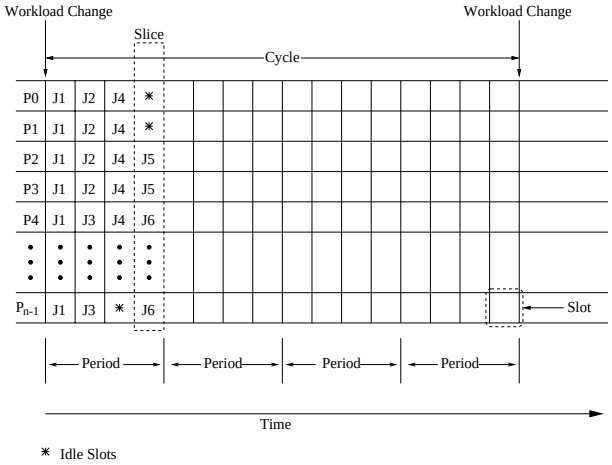

Figure 1: Cycle, slice, period and slot definitions

change again occurs. In the event of a workload change, the distribution of jobs in the machine is reorganized depending on the change in the workload, and as we have a queue of jobs, some thread migration may occur because of this reorganization. We will refer to this strategy henceforth simply as first fit.

Although we defined an algorithm where thread migration was possible, if the machine under consideration has no efficient mechanism for thread migration, algorithms with no thread migration are also possible using these concepts.

A very simple policy for spatial sharing under Concurrent Gang without thread migration is the greedy one. At arrival, a job is scheduled in a slice that has sufficient idle slots to accommodate the arriving job. In this case the definitions of cycle, slice, etc. would also be valid. The scheduler should maintain a list of idle slots in the period in order to know, at job arrival, if it is possible to schedule the job in an already existing slice.

It is worth noting that, relative to its definition as a queueing network with processor sharing discipline, Concurrent Gang is particularly convenient to describe schedulers that are periodic between workload changes. We will now state a theorem that proves that a periodic schedule performs at least as well as any non periodic one with respect to the total number of idle slots, i.e., periodic schedulers achieves better spatial allocation than (or at least as good as) non-periodic ones when processor utilization is measured through the ratio of total number of

empty (idle) slots to the total number of slots in the period. We denote this measure as the *idling ratio.*

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 make 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 infinitely in the time dimension.

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.

4 Simulation and Verification

To verify the results above, we used a general purpose event driven simulator, developed by our research group for studying a variety of related problems (e.g., dynamic scheduling, load balancing, etc.). The simulator accepts two different formats for describing jobs. The first is a fully qualified DAG. The second is a set of parameters used to describe the job characteristics such as computation/communication ratio.

When the second form is used the actual communication type, timing and pattern are left unspecified and it is up to the simulator to convert this user specification into a DAG, using probabilistic distributions, provided by the user, for each of the parameters. Other parameters include the spawning factor for each thread, a thread life span, synchronization pattern, degree of parallelism (maximum number of threads that can be executed at any given time), depth of critical path, etc. 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 heuristics is plugged into the simulator, several experiments can be performed using the same input by changing some of the pa-

rameters 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.

For this study we grouped parallel jobs in classes where each class represents a particular degree of parallelism (maximum number of threads that can be executed at any given time). The reason behind grouping parallel jobs by their degree of parallelism is to evaluate the performance of the algorithms being studied across the vast spectrum of real parallel applications (ranging from massive parallel to programs requiring only two processing elements) and therefore reduce the bias towards a single type of application.

We divided the workload into ten different classes with each class containing 50 different jobs. The arrival time of a job is described by a Poisson random variable with an average rate of two job arrivals per time slice. The actual job selection is done in a round robin fashion by picking one job per class. This way we guarantee the interleaving of heavily parallel jobs with shorter ones.

We distinguish the class of computation instruction and that of communication instruction in the various threads that compose a job. The latter forces the thread to be suspended until the communication is concluded. If the communication is concluded during the currently assigned time-slice the thread resumes execution. We used a factor of 0.001 communications per computation instructions.

The classes are ranked according to their degree of parallelism (between 2 and 1024 in powers of two increments) and the jobs were scheduled in a simulated 1024 processor machine. In table 1 we compare gang scheduling with Concurrent Gang using first fit as space sharing strategy.

It is important to dissect the value obtained for idle time, which is the result of three factors:

1 - Communications

2 - Absence of ready threads

3 - Inneficiency of allocation

The first is a natural consequence of threads communicating among themselves. The second reflects the fact that jobs arrive and finish at random times and at any given instance there might not be any job ready to be scheduled. The last is a result of inefficiencies due to the non optimality of the first fit algorithm.

Gang	
Total Running Time (%)	Total Idle Time (%)
123.6	41.9
Concurrent Gang	
Total Running Time (%)	Total Idle Time (%)
100	28.2

Table 1: Experimental results

References

[1] Feitelson, D. G.: Job Scheduling in Multiprogrammed Parallel Systems *IBM Research Report RC 19970*, Second Revision, 1997

[2] Feitelson, D. G., Jette, M. A.: Improved Utilization and Responsiveness with Gang Scheduling *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238-261 Springer Verlag, 1997.

[3] Feitelson, D. G.: Packing Schemes for Gang Scheduling *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89-110 Springer Verlag, 1996.

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

[5] Jette, M. A., Performance Characteristics of Gang Scheduling in Multiprogrammed Environments, *Supercomputing' 97*, 1997.

[6] Scherson, I. D., Subramaniam R., Reis, V. L. M., Campos, L. M.. Scheduling Computationally Intensive Data Parallel Programs. *Ecole Placement Dynamique et Re'partition de charge* pp. 39-61, 1996

[7] Silva,F., Campos, L. M., Scherson, I. D. A Lower Bound for Dynamic Scheduling of Data Parallel Programs, *EUROPAR' 98* (1998 - to appear)