Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 00-014

Gang Scheduling for Distributed Memory Systems
William Leinberger, George Karypis, and Vipin Kumar

February 16, 2000
Gang Scheduling for Distributed Memory Systems *

William Leinberger, George Karypis, and Vipin Kumar

(leinberg, karypis, kumar)@cs.umn.edu

Army High Performance Computing Center, Minneapolis, MN
Department of Computer Science and Engineering, University of Minnesota

February 16, 2000

Abstract

The addition of time-slicing to space-shared gang scheduling improves the average response time of the jobs in a typical job stream. Recent research has shown that time-slicing is most effective when the jobs admitted for execution fit entirely into physical memory. The question is, how to select and map jobs to make the best use of the available physical memory. Specifically, the achievable degree of multiprogramming is limited by the memory requirements, or physical memory pressure, of the admitted jobs. We investigate two techniques for improving the performance of gang scheduling in the presence of memory pressure: 1) a novel backfill approach which improves memory utilization, and 2) an adaptive multiprogramming level which balances processor/memory utilization with job response time performance. Our simulations show that these techniques reduce the average wait time and slow-down performance metrics over naive first-come-first-serve methods on a distributed memory parallel system.

1 Introduction

Classical job scheduling strategies for parallel supercomputers have centered around space-sharing or processor sharing methods. A parallel job was allocated a set of processors for exclusive use until it finished executing. Furthermore, the job was allocated a sufficient number of processors so that all the threads could execute simultaneously (or gang'ed) to avoid blocking while attempting to synchronize or communicate with a thread that had been swapped out. The typical poor performance of demand-paged virtual memory systems caused a similar problem when a thread was blocked due to a page fault. This scenario dictated that the entire address space of the executing job must be resident in physical memory \cite{2,4,9}. Essentially, physical memory must also be ganged with the processors allocated to a job. While space-sharing provided high execution rates, it suffered from two major drawbacks. First, space-sharing resulted in lower processor utilization due to some processors being left idle. These processor "holes" occurred when there were not sufficient processors remaining to execute any of the waiting jobs. Second, jobs had to wait in the input queue until a sufficient number of processors were freed up by a finishing job. In particular, many small jobs (jobs with small processor requirements) or short jobs (jobs with short execution times) may have had to wait for a single large job with a long execution time, which severely impacted their average response time.

Time-slicing on parallel supercomputers allows the processing resources to be shared between competing jobs. Each parallel job is gang-scheduled on its physical processors for a limited time quantum, TQ. At the end of the time quantum, the job is swapped out and the next job is swapped in for its TQ. This improves overall system utilization as processors which are idle during one time quantum may be used during a different time quantum. Gains in response time are also achieved, since the small and short jobs may be

*This work was supported by NASA NCC2-5268 and by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003 (contract number DAAH04-95-C-0008). Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute.
time-sliced with larger, longer running jobs. The larger jobs make progress, while the small jobs are not blocked for long periods of time [5, 10].

Time-slicing uses a maximum multi-programming level (MPL) to control the number of jobs mapped to the same physical processors. Once mapped to a set of physical processors, the jobs’ execution rate is dependent on the number of other jobs mapped to the same processors, or the current MPL. The slower execution rate of the time-sliced jobs is offset by a reduced waiting time and higher system utilization. In general, for job streams with a high percentage of small or short jobs, the benefits of time-slicing increase with increasing MPL. Herein lies the primary impact of memory considerations to time-sliced gang scheduling. The achievable MPL is limited by the need to have the entire address space of all admitted jobs resident in physical memory. Therefore, effective time-slicing requires efficient memory allocation in order to maximize the achievable MPL. Additionally, the current benefits of high processor utilization and job response time must be maintained [1, 11].

Our first contribution is a novel approach for selecting jobs and mapping them to empty slots through a technique which we call weighted memory balancing. Job/slot pairs are selected which maximize the memory utilization, resulting in a higher achievable MPL. However, aggressive memory packing can lead to fragmentation in both the memory and the processors, causing delays to jobs with large resource requirements. Therefore, our second contribution is to provide an adaptive multi-programming level heuristic which balances the aggressive memory packing of small jobs with the progress requirements of large jobs.

The remainder of this paper is as follows. Section 2 provides an overview of the state-of-the-art in time-sliced gang scheduling (referred to as simple gang scheduling in current literature). We describe our new memory-conscious techniques in Section 3. We also describe their integration into current methods. Section 4 describes a simulation exercise that we used to evaluate our techniques on synthetic job streams. Included in this section is our model for the dual-resource job stream. Section 5 concludes with a discussion of our work-in-progress.

2 Preliminaries

2.1 System Model

Our parallel system model is a distributed memory parallel processor of size P. Each processor has a local memory of size M. All processors are interconnected tightly enough for efficient execution of parallel applications. This model encompasses both the parallel supercomputers, like the IBM SP2 or Intel Paragon, as well as the emerging Beowulf class PC super-clusters. Examples supporting this system model are the IBM ASCI-Blue system [8], and the ParPar cluster system [1]. We also assume that there is no hardware support for parallel context switching. Time-slicing parallel applications on these types systems is generally conducted using a coarse, or large, time quantum to amortize the cost of context switching between jobs. This is especially relevant since the parallel context switching is implemented in software.

2.2 Job Selection and Mapping for Gang Scheduling

Gang scheduling begins with selecting a job for execution and then mapping the job on to the available resources. The selection of the next job for mapping is either first-come-first-serve (FCFS), or a re-ordering of the input queue. FCFS suffers from blocking small jobs in the event that the next job in line is too large for any of the open slots. Backfilling has commonly been used in space-shared systems for reducing the effects of head-of-line (HOL) blocking by moving small jobs ahead of large ones [7]. EASY backfilling constrains this re-ordering to selecting jobs which do not interfere with the earliest predicted start-time of the blocked HOL job. This requires that the execution times of waiting jobs and the finishing times of executing jobs must be calculated. In a time-sliced environment, an approximation to the execution time of a job is the predicted time of the job on an idle machine times the multi-programming level [12]. Various policies for re-ordering have been studied such as first-fit which selects the next job in the ready queue for which there is an available slot, and best-fit which selects the largest job in the ready queue for which there is an available slot.
Given a \( p \)-processor job, the gang scheduler must find an empty time slot in which at least \( p \) processors are available. While many methods have been investigated for performing this mapping [3], the Distributed Hierarchical Control (DHC) method achieves consistently good performance [6]. We use DHC as our baseline mapping method. The DHC method uses a hierarchy of controllers organized in a buddy-system to map a \( p \)-processor parallel job to a processor block of size \( 2^{\left\lfloor \log_2(p) \right\rfloor} \). A controller at level \( i \) controls a block of \( 2^i \) processors. The parent controller controls the same block plus an adjacent "buddy" block of the same size. A \( p \)-processor job is mapped to the controller at level \( i = \left\lfloor \log_2(p) \right\rfloor \) which has the lightest load. This results in balancing the load across all processors. A maximum MPL is used to limit the number of jobs mapped to any single processor.

2.3 Gang Scheduling with Memory Considerations

Research into the inclusion of memory considerations into gang scheduling is just beginning. Batat and Feitelson [1] investigated the trade between mapping a job to a slot and relying on demand-paging to deal with memory pressure vs. queueing the job until sufficient physical memory was available. For the system and job stream tested, it was better to queue the job. Setia, Squillante, and Naik investigated various re-ordering methods gang scheduling a job trace from the LLNL ASCI supercomputer [11]. One result of this is that restricting the multi-programming level can impact the response time of interactive jobs as the batch jobs dominate the processing resources. They proposed a second parameter which limited the number of large batch jobs into the system.

3 Memory Management Techniques for Gang Scheduling

We provide new techniques for job selection and mapping which improve memory utilization in a time-sliced gang scheduling system. These techniques are then integrated with the concepts described in Section 2. We use the DHC approach to control the local scheduling of jobs to time slots. However, we integrate the job selection and mapping process so that we can select a job/slot pair based on memory considerations. We provide an intelligent job/slot selection algorithm which is based on balancing the memory usage among all processors, much like the DHC balances the load across all processors. This is described below in Section 3.1. This job/slot selection algorithm is based on EASY backfilling, which is subject to blocking the large jobs [12]. We also provide an adaptive multi-programming level which is used to control the aggressiveness of the intelligent backfilling. This adaptive MPL is described below in Section 3.2.

3.1 Memory Balancing

Mapping a job mapping based on processor loading alone can lead to a system state where the memory usage is very high on some processors but very low on others. This makes it harder to find a contiguous block of processors with sufficient memory for mapping the next job. We refer to this condition as pre-mature memory depletion. Figure 1 shows a distributed memory system with \( P=8 \) and MPL=3. Figure 1 (a) depicts the result of mapping a job, \( J_0 \) (described by processor and memory requirements \( P_0 = 1 \) and \( M_0 \) respectively), by a level 0 controller to processor 0 using a first-fit approach. This results in depleting most of the memory on processor 0, making it difficult for this controller to map additional jobs to the remaining time slot. The parent controller, which maps jobs with a 2-processor requirement to processors 0 and 1, will also be limited. In general, all ancestor controllers will be limited by this mapping.

An alternative mapping is depicted in Figure 1 (b). Here, \( J_0 \) is mapped to processor 4, leaving more memory available for the ancestral controllers to place future jobs. Processor 4 was selected because it had the lowest load (one thread) and left the most memory available for other jobs. One heuristic for achieving this is as follows. Map the job to the controller which results balancing the memory utilization across all the processors. We define the memory balance as \( \text{Max}(M_i)/\text{Avg}(M_i), 0 \leq i < P \).

This notion can be further refined by noting that mapping a job to a controller on level \( i \) directly affects the memory on the \( 2^i \) processors managed by that controller, so we first measure the balance across these processors. This local balance score is weighted by the probability that a job of size \( 2^{i-1} < p < 2^i \) arrives in the future. Essentially, the weighted balance score measures the ability of the controller to meet the memory
requirements of future jobs on its processors. Continuing, measure the balance across the $2^{k+1}$ processors controlled by the parent of this controller, and so on, until we measure the balance across the entire range of processors in the system. As we progress up the levels of controllers, the balance measured at each controller is weighted by the probability of needing a slot of the size managed by that controller. The total score is the average of the weighted scores at each level.

We implemented the above heuristics in our Weighted Balance (WBAL) algorithm in our simulated system as described below in Section 4. The WBAL algorithm uses EASY backfilling [12], but uses the balance score to select the next backfill job instead of first-fit. The job size distributions are derived by keeping track of the sizes of jobs which have been previously scheduled. During a scheduling epoch, we score each job in the ready queue for against each appropriate slot. The job/slot pair with the best score is selected for admission.

### 3.2 Adaptive Multi-Programming Level

Aggressive backfilling methods move smaller jobs ahead of larger blocked jobs in an effort to improve average job response time. The backfill jobs may not interfere with the predicted start time for the job blocked at head of the Ready Queue, RQ. However, the job which is next-in-line in the queue may be delayed severely by the backfilling due to space fragmentation. Consider the system state depicted in Figure 2 (a). The jobs are numbered according to their arrival, with $J_{10}$ arriving before $J_{11}$, etc.. At some point, job $J_{14}$ was delayed at the head of the queue, with $J_{15}$ right behind it. $J_{16}$ and $J_{17}$ are backfilled, since neither interferes with the earliest start time for $J_{14}$. However, $J_{17}$ inadvertently delays $J_{15}$. This space fragmentation effect compounds against the third, fourth, etc., jobs waiting at the head of the ready queue. Although the small jobs are moved ahead, the large jobs may be delayed a disproportionately long time.

Figure 2 (b) depicts the average slow-down performance of a workload in which the average memory requirement is 25% of the available physical memory. For MPL $\leq$ 4, the performance increases with increasing MPL. However for MPL $> 4$, the performance decreases, due to over-aggressive backfilling. We developed a heuristic to adaptively adjust the maximum MPL, based on the natural level dictated by the waiting jobs. The natural level is the total memory per processor, M, divided by the average per-processor memory requirement of the jobs waiting in the queue. If the backfilling is temporarily achieving a multi-programming level above this natural level, then a lot of jobs with small memory requirements are being selected in favor of the jobs with larger memory requirements. Periodically, the maximum MPL is re-calculated as
$M/Avg(M_i), J_i \in RQ$. This heuristic was added to the simulated scheduling algorithms in Section 4 as the Adaptive Multi-programming level technique, or AM.

4 Experimental Results

In this section, we describe the simulation methods used to evaluate the new techniques. Our our dual-resource workload model is described in Section 4.1, and our simulation results are presented in Section 4.2.

4.1 Workload Model

Past research on workload models has focused on a single resource, processors [3]. The results of these efforts generally provide that processor requirements follow a hyperexponential distribution, with many strong discrete components at powers of two and squares of integers. Recent efforts is characterizing memory usage show that memory and processor requirements are weakly correlated [4, 1]. Also, the size of the memory requirements, while high, still allow for a low degree of multi-programming in physical memory [11]. We generalize this conclusion with the following dual-resource workload model. First, the probability distributions for the processor and memory resource requirements are generated with a specific mean, $R_A$, and variance, $R_V$. Example distributions are depicted in Figure 3 (a). The requirements for a given job are then drawn from the respective distributions in such a way as to create a job stream in which the processor and memory requirements are correlated as specified by a resource correlation parameter, $R_C$. Histograms for two values of $R_C$ are depicted in Figure 3 (b). The X and Y axis represent the various values of P and M, while the Z axis is the number of times that combination of P and M were generated in job. The execution times for jobs are drawn from a hyperexponential distribution as well. The inter-arrival times are exponential. System load is adjusted by changing the inter-arrival rate.

4.2 Simulation Results

We implemented the DHC time-sliced gang scheduler on a simulated system, with three different job-selection/mapping algorithms. The first job selection algorithm is the first-come-first-serve (FCFS), which takes jobs from the head of the queue and places them onto the least loaded slot of the appropriate size, with sufficient physical memory. Second, the FCFS was modified to include EASY backfilling (EASY). Finally, the WBAL algorithm as described in Section 3.1 was implemented. The EASY and WBAL algorithms were also simulated with the adaptive MPL, and are denoted as EASY/AM and WBAL/AM in the figures. We assume perfect knowledge for resource requirements and execution time.

The simulated parallel system used P=64 and M=64. The algorithms were evaluated on the basis of the average slow-down performance metric. The slow-down metric is the ratio of the execution time on the loaded machine (wait time plus reduced execution rate) to the execution time on an idle machine (no waiting, full execution rate). We used a single distribution for the processor requirements with the $R_A^P = 1/8$, or roughly
8 of the 64 available processors. We used two different memory requirement distributions, $R_A^M = 1/4$, and $R_A^M = 1/3$s. Results are also reported for three different resource correlation values, $R_C = +0.7$, $R_C = 0.0$, and $R_C = -0.7$. The average slow-down performance results are depicted in Figure 4. Figure 4 (a) depicts results for $R_A^M = 1/4$ over all three values for $R_C$ and (b) depicts similar results for $R_A^M = 1/3$. Overall, the WBAL/AM consistently performs as good or better than the EASY/AM and EASY. Additionally, EASY/AM performs as good or better than EASY.

At lower memory pressure, Figure 4 (a), the backfill based algorithms perform about the same, with WBAL/AM slightly better than EASY/AM and EASY. When the processor and memory requirements are negatively correlated, EASY/AM and WBAL/AM perform much better than EASY. In this case, the many small processor jobs generally have high memory requirements. The combination of un-restrained MPL and the first-fit mapping used by EASY results in pre-mature memory depletion. EASY/AM and WBAL/AM perform well due to the fact that the multi-programming level is naturally small, as jobs have high per-processor memory requirements. The AM heuristic prevents EASY/AM and WBAL/AM from aggressive backfilling.

At higher memory pressure, Figure 4 (b), and higher resource correlation, the improved packing efficiency produced by WBAL and moderated by AM results in WBAL/AM achieving better performance than the EASY backfill variants. This is significant as these workloads correspond most closely to the findings of the studies on memory requirements for scientific workloads, described earlier. Encouraged by the results, we are currently working towards applying our techniques to real dual-resource workloads. As the resource correlation goes negative, the performance of EASY/AM and WBAL/AM increases over EASY, as a result of the AM heuristic preventing overly aggressive backfilling.

5 Summary and Future Work

The combination of the weighted memory balancing and the adaptive multi-programming level heuristics produced a job selection and mapping algorithm, WBAL/AM, which consistently outperformed other methods. However, these methods are backfill based, requiring a priori knowledge of resource requirements and execution times. Our current work in progress is aimed at overcoming this limitation. The basic idea is that jobs are selected and mapped, using information that may initially have errors. However, once jobs have executed for a few time slices, resource requirement information is improved. In the event that memory becomes over-subscribed this can be used to decide which jobs to swap out of memory, in the classical sense, to disk.

References

Figure 4: Average Slow-Down Results


