2✓

CONF-8911132--1

UCRL- 102059
PREPRINT

SIMULATING THE SCHEDULING OF PARALLEL SUPERCOMPUTER APPLICATIONS

Mark K. Seager
James M. Stichnoth

User Systems Division
Lawrence Livermore National Laboratory

This paper was prepared for submittal to
IBM - JT
Watson Center
Yorktown Heights, NY
November 2-3, 1989

September 19, 1989

Lawrence Livermore National Laboratory

Simulating the Scheduling of Parallel Supercomputer Applications [†]

UCRL--102059

DE90 003604

Mark K. Seager
James M. Stichnoth
User Systems Division
Lawrence Livermore National Laboratory

September 19, 1989

Abstract

An Event Driven Simulator for Evaluating Multiprocessing Scheduling (EDSEMS) disciplines is presented. The simulator is made up of three components: 1) machine model; 2) parallel workload characterization and 3) scheduling disciplines for mapping parallel applications (many processes cooperating on the same computation) onto processors. A detailed description of how the simulator is constructed, how to use it and how to interpret the output is also given. Initial results are presented from the simulation of parallel supercomputer workloads using "Dog-Eat-Dog," "Family" and "Gang" scheduling disciplines. These results indicate that Gang scheduling is far better at giving the number of processors a job requests than Dog-Eat-Dog or Family scheduling. In addition, the system throughput and turnaround time are not adversely affected by this strategy.

[†]Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

1 Introduction

With the advent of parallel Multiple Instruction Multiple Data (MIMD) shared memory computer systems, the need to support parallel programming has arisen. This need is expressed at many levels including compiler, loader, library support and the operating system. At the compiler level the most successful automatic support for parallel programming is microtasking or autotasking [2]. Other strategies for parallel programming usually include some form of user directed parallelism or multitasking (c.f., [3] or [7]) and require additional language key words (e.g., `task common`, `shared` and `private`) for shared and/or private data as well as library support for creating and managing tasks or processes (e.g., `fork`, `tskstart`, locks and barriers). At the loader level data shared between parallel work sections (tasks or processes) must be put in a shared address space and data which is private to a task or process must be replicated. At the operating system level there must be some way to have processes share data (address spaces). On all MIMD shared memory systems that we are aware of, some or all of the above support for parallel programming is supplied. It is amazing that all of these systems fail to support parallel multiprogramming in a very fundamental way. They all have process scheduling policies and algorithms that were designed for serial computers and simply ported to the new parallel environment. Hence, it is not uncommon for parallel applications to have severe problems on multiprogrammed parallel systems. Some typical problems are:

- Excessive spin waiting. On systems such as Sequent [7] all locks can, at the discretion of the developer, either be implemented with spin-waiting or spin waiting with blocking. For reduced overhead in parallel programming, most users opt for spin waiting without the blocking. This strategy works fine when there are enough idle CPUs to satisfy the parallel program's requirements. On the other hand, when multiple parallel programs are scheduled simultaneously so that at least one process from some of the jobs are not running, the system tends to "lock-up" quickly (due to spin-wait barrier synchronization) and most cycles are spent spin-waiting until the end of process' timeslice.

- Poor average parallel overlap. On a moderate to heavily loaded system environment, such as that found on most supercomputers, parallel applications use synchronization with blocking. This relieves the problem of excessive spin waiting, but each process of parallel applications competes with every other process in the system (based on priority, amount of service, etc.) for allocation to the next available CPU. Hence, there is a high probability that only a few of the processes of the parallel application will actually be running at the same time. This can result (depending on the workload) in high context switching rates for the system as a whole and poor average parallel overlap for the individual application.

When faced with the problem of reduced system throughput and abysmal performance of parallel applications in multiprogramming environments, most system administrators decide that running multiple serial job streams on parallel supercomputers is the most efficient means of machine utilization. Applications developers take the view that, under these conditions, they would be better off spending their time working on developing "new physics" (i.e., improving the simulations) than battling with the problems associated with parallel programming.

This is the situation facing most supercomputer sites today. It is not clear that the "natural" conclusion reached by system administrators and developers alike to forgo parallelism within applications is indeed the correct one. In order to counter these religious arguments[1] for multistreaming parallel supercomputers with facts, but not incur the cost and inconvenience of experimentation with the "real system," a simulator was developed to study scheduling algorithms on MIMD shared memory computers.

This paper describes the Event Driven Simulator for Evaluation of Multiprocessing Scheduling algorithms (EDSEMS). It was developed to study various scheduling algorithms on shared memory multiprocessors with various workload models. Section 2 describes the simulator from a user's perspective and how to set up an input file. Section 6 describes the EDSEMS interactive mode and the batch mode's output structure. Section 3 describes the scheduling methods implemented in EDSEMS. Section 4 describes the unique workload model used in the simulator. Section 5 discusses the implementation details

[1]Arguments based on assumptions and preconceived notions which have not been subject to the scientific method: verification.

for the simulator. Section 7 gives results for some scheduling/workload studies done with EDSEMS and discusses the ramifications.

For the reader wishing to use the EDSEMS simulator the following sequence is recommended: $1 \Rightarrow 2 \Rightarrow 6 \Rightarrow$ A. Readers interested in modifying the simulator should study: $1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5$. Readers interested in early simulation results are directed to: $1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 7 \Rightarrow 8$.

2 User Guide

To run the simulator one needs the EDSEMS executable program, and some input files. (Input files are not necessary if the default values for the input parameters are acceptable.) The simulator can be run in interactive or batch mode. Command line switches determine the mode of the run. In batch mode the simulation is run and the output of the initial parameters, short-term and long-term statistics are printed to `standard out`. Command line options include `-h`, which turns off the printing of the histograms (see below) and `-d Debug_Level` which overrides the value of the `DEBUG` input value (see below).

The remaining flags `-s` and -w JID, if present, put the simulator into interactive mode. In interactive mode, the processes running, as well as the contents of the queues, are displayed via the Curses terminal interaction library [1]. In this mode short-term and long-term statistics are not printed. To activate interactive mode one specifies at least one job the user would like to monitor with the `-w JID` command line option. This specifies that job number `JID` should be displayed in reverse video while it is active in the system. Up to ten jobs can be monitored in a single run.

The `-s` flag overrides the `-w` flag. This flag causes processes which are spin waiting, blocked, doing context switching overhead, or otherwise non-runnable, to be displayed in reverse video. This allows the user to monitor, in particular, the amount of time the CPUs spend *not* doing user work.

In either mode optional input files can be specified on the command line. Multiple input files are allowed, and the files are parsed in the order they appear on the command line. This allows variables set in previous files to be overridden and is useful when making many similar runs in batch mode, since smaller input files can be used.

Standard input is used if no argument is given, or if an argument is "-". Although all simulation parameters have default values, the input files are used to override these defaults. Parameters that can be modified include: length of the simulation, delta simulated time for short-term outputs, system time slice interval, system overhead in process exchange, average number of barriers per job, average work between barriers, and distribution of number of processes per job. Many parameters are given as means and standard deviations, so that the user can regulate the types of jobs being simulated.

2.1 Input File Structure

The input file format is fairly flexible. Statements can appear in any order and comments can appear on any line. Comments start with "#" and continue to the end of the line. Input statement lines are of the format ⟨*identifier*⟩ = ⟨*value*⟩. If an identifier is not given a value in the input file, it is set to a default. Although the input parameters are dimensionless (i.e., they are all relative) it is ueful to think of the default parameters being given in micro-seconds (1.0E-6 seconds). Identifiers in the input file must match one from the following list (including case):

- `DEBUG` (default = 0): Debugging output flag. For each level of `DEBUG`, the output from that level and lower levels are printed. The levels print the following information:

  0 No debugging output is printed.

  1 Simulation time-timestamp for each job creation and completion. At job creation, all parameters of the job are printed (e.g., number of barriers to execute and mean time between barriers).

  2 Same as 1.

  3 Per process statistics every time a process completes.

  4 All state changes in the system are reported. This is very detailed debugging output and generates a tremendous amount of output.

- `SchMethod` (default = 0): Scheduling method. 0 means "Dog-Eat-Dog" scheduling, 1 means "Family" scheduling and 2 means "Gang" scheduling. See Section 3 for a detailed description of the scheduling methods.
- `GenMethod` (default = 0): Job generation method. See Section 4 for a detailed exposition of the job generation methods. The following values are mutually exclusive:
  - 0 "Min" process job generation method. The simulator tries to keep at least `MinProcsInSystem` processes in the system at all times.
  - 1 An "Exponential" distribution with mean `DelayMean` is used to generate the wait time between jobs. This method is useful when trying to model "typical" systems with loads that vary with time.
  - 2 "Load" process job generation method. The simulator generates a new job when the system "load" is below the user specified minimum `MinLoad`. This is useful when one is trying to keep the system at a constant busy level. The "Min" method does not keep the system loaded in a reproducible way because processes block randomly and cause the load to vary.
- `NumCPUs` (default = 8): Number of CPUs in the simulated system.
- `MinProcsInSystem` (default = 10): Minimum number of processes in the system at one time. This parameter only has meaning when using the "Min" job generation method.
- `MinLoad` (default = 3.0): Minimum load to keep running in the system. This parameter only has meaning when using the "Load" job generation method.
- `DelayMean` (default = 500000.0): Mean time between job arrivals. This parameter only has meaning when using the "Exponential" job generator.
- `SIMMean` (default = 1350.0) and `SIMStdDev` (default = 135.0): At job creation time, each job is given a mean work between barriers, λ, statistic from an normal distribution with mean `SIMMean` and standard deviation `SIMStdDev`. λ is used in an exponential distribution to determine the "base work time" before the next barrier for all processes within the job. (The "SI" stands for "Synchronization Interval", which is the work time between barriers.)
- `SISMean` (default = 135.0) and `SISStdDev` (default = 13.5): From a normal distribution with mean `SISMean` and standard deviation `SISStdDev` we choose, at job creation time, a standard deviation, σ, for the noise each process will add to the job-global work between barriers. During the simulation, each process determines the work time to consume before the next barrier by adding "barrier noise" to the job-global work between barriers computed from an exponential distribution (see `SIMMean` and `SIMStdDev`, above). The noise is a normal random variable with mean zero and standard deviation σ. Each process adds "noise" to the job-global base work time (see `SIMMean` and `SIMStdDev`, above) to simulate load imbalance in the application code. (The "SI" stands for "Synchronization Interval," which is the work time between barriers.)
- `NBMean` (default = 1000.0) and `NBStdDev` (default 100.0): At job creation time, the number of barriers a job is to perform is determined from a normal distribution with mean `NBMean` and standard deviation `NBStdDev`. On average, the total runtime of a job is then the mean number of barriers (`NBMean`) times the mean work between barriers (`SIMMean`).
- `GlobalTimeSlice` (default = 100000.0): This is the time slice that each process gets. When its time slice runs out, it is preempted.
- `GlobalSpinWaitDelay` (default = 1000.0): When a process reaches a barrier, it will hold onto its CPU and spin wait, waiting for the other processes in the job to reach the barrier, at which point the process will continue. If the process has not been reactivated within this amount of time, the process will voluntarily give up its CPU.

- `GlobalOverhead` (default = 350.0): This is the amount of time it takes for the scheduler to complete a context switch.
- `SimLength` (default = 1.0E6): This is the length, in simulated time, of the simulation.
- `OutputDelta` (default = 50000.0): This is the simulation time between the printing of statistics.
- `RandomSeed` (default = 0): If `RandomSeed` is negative, the random number generator is not seeded, thus producing a repeatable sequence. If `RandomSeed` is positive, the random number generator is seeded with that value, also producing a repeatable sequence. If `RandomSeed` is 0, the random number generator is seeded according to the time, thus producing a different sequence of random numbers for each run.
- `ParArray`: This array describes the distribution of the number of processes per job. The values should be all on one line, separated by whitespace. There must be `NumCPUs` values, and the sum of all the values must be 1. The nth value of the array represents the probability that the next job generated will contain n processes. The default is to give equal probability for all values.

Periodically the simulator will output short-term and long-term statistics. These statistics include: number of jobs completed, breakdown of time spent for the jobs (i.e. run time, idle time, blocked time), level of parallel overlap, CPU utilization, load average and other items. The output also includes histograms showing processes per job, percentage of spin wait time used by each process, and percentage of time slice used by each process. The output will be discussed in more detail in Section 6.

While the simulation is running, the user can interagate the simulator to determine the progress of the simulation. This is accomplished with the Unix QUIT interrupt (usually mapped to the `CTRL-\` character. EDSEMSresponds to the Unix QUIT interrupt by printing the current simulated time, the amount of CPU time used by the simulator and the ratio of the two to the terminal. In interactive mode, this is displayed on the bottom line of the screen.

3 Scheduling Algorithms

Three scheduling disciplines are currently implemented in the EDSEMS simulator: Dog-Eat-Dog, Family and Gang. Dog-Eat-Dog coresponds to placing a scheduling method developed for a serial architecture on one with multiple processors. Family coresponds to updating Dog-Eat-Dog so that it knows something about jobs, but does not take an pro-active role in obtaining processors for parallel jobs. Gang scheduling coresponds to running jobs with as many CPUs as they request (if at all possible). The following describes each scheduling method in more detail.

3.1 Dog-Eat-Dog

We called the first method "Dog-Eat-Dog," because in this discipline, each process in the system competes with every other process for the CPUs, without any cooperation between processes in the same job. Dog-Eat-Dog is just a textbook implementation of preemptive Round-Robin scheduling [6] on a parallel architecture.

Today, most computer vendors selling Multiple Instruction Multiple Data (MIMD) parallel architectures (e.g., Sequent, Alliant, Cray, Convex) have simply opted to port process scheduling algorithms from operating systems designed for serial machines. Consequently, the scheduling discipline has no knowledge of processes that may cooperate in a parallel computation. These vendors end up with some variant of Dog-Eat-Dog scheduling.

Dog-Eat-Dog is implemented in the simulator by having one ready queue in the system. Processes enter the ready queue at the rear. When a CPU becomes available, it is given to the process at the front of the ready queue. When that process blocks (due to synchronization) or its time slice runs out, it gives up its CPU and goes to the rear of the ready queue, and the CPU is assigned to the next ready process.

In the EDSEMS implementation, we chose not to have a blocked list. Instead, blocked processes are also kept on the ready queue. When a CPU is available and a blocked process is at the front of the ready queue, that process is just moved to the back of the queue, and the next ready process is taken. This coupled with the fact that processes from parallel jobs come in as a group, imply that processes from a parallel job are contiguous in the ready queue. Hence, Dog-Eat-Dog tends to schedule parallel jobs in a manor similar to, but not exactly like, Family scheduling.

3.2 Family

In an attempt to improve the "co-scheduling" of processes from a parallel job, a second method called "Family" scheduling was developed at LLNL for use in the Networked Livermore TimeSharing operating System (NLTSS) [5], [10] on Cray X-MP and Y-MP class parallel MIMD supercomputers. A "family" is just a job, and processes in the same job are called members. Family scheduling can be summarized by the statement: "When, in the natural course of events, a family member is scheduled, all other runnable (non-blocked) members in that family are scheduled **as CPUs become available**." This statement clearly summarizes Family scheduling when there is only one (or fewer) parallel job running at a time (a common event in the current workload at LLNL [8]). What to do when there are multiple parallel jobs wanting to run is a bit more complicated and is described below.

Family scheduling has knowledge of jobs as well as processes and attempts to increase the probability that family members run simultaneously. The goal is to improve the level of parallel overlap, decrease spin wait time and blocked queue latency, thus improving system throughput. Family scheduling also has the feature that it does not take away a process' CPU unless the process' time slice runs out or the process blocks. Consequently, this discipline is very easy to defend politically, because it causes no disruption of service for other jobs that happen to be running contemporaneously with the family.

In order to account for the fact that many families may be in the system at any one time, we have opted for two ready queues and a blocked list. One queue is a high-priority queue (HPQ) and the other is a low-priority queue (LPQ). Processes start out on the LPQ. When a CPU becomes available, the scheduler assigns the CPU to the first process on the HPQ. If the HPQ is empty, the scheduler assigns the CPU to the first process on the LPQ, and moves all of its family members to the HPQ. When these processes are moved to the HPQ, and there are already other family members in the HPQ, then the remaining family members are placed after the last family member in the HPQ; otherwise, the family members are placed at the front of the HPQ. When a process blocks, it goes onto the blocked list, and when it unblocks, it goes onto the HPQ. If a process is the only family member who has a CPU, and that process gives up its CPU, then all family members on the HPQ are moved to the rear of the LPQ.

3.3 Gang

A third method that implemented in EDSEMS for scheduling processes on a parallel architecture is "Gang" scheduling. In this discipline we take the attitude that CPUs are like non-virtual memory on a Cray[2]: either you get what you request or you don't. On Cray class supercomputers you either fit into memory or you don't. The available memory changes dynamically, depending on what jobs are currently resident in memory. Generally, a large memory job will be swapped out to disk for longer periods of time than a shorter job, but when it is rolled into memory it is given a longer memory "residency time." In Gang scheduling, we tried to model this approach to resource allocation by giving jobs (as far as possible, see the discussion of floaters, below) the CPUs they request or none at all. The more CPUs a job requests, the longer (in general) it will have to wait to get them.

The goal of Gang scheduling is to deliver the largest possible parallel overlap (average parallelism delivered to an application) without idling the hardware. This is advantageous for several reasons:

1. The maximum speedup an application can obtain (in terms of run time, not turnaround time) is the average overlap of processes from the job delivered by the operating system.

2. Due to Amdahl's law, it is quite difficult to maintain a constant efficiency (speed-up divided by number of CPUs used) as one adds more and more parallelism. Hence, if a developer goes to the work of parallelizing his application to level 8, say, and only gets an observed overlap of 2.5

[2]Actually, on supercomputers available today, memory and memory-to-CPU interconnects are by far the most expensive part of the machine! This is in stark contrast to the situation of a decade ago.

(not uncommon running on a Cray Y-MP/832 with NLTSS or UNICOS operating systems), he is paying the efficiency penalty of eight way parallelism, but actualizing only 2.5 way parallelism. Thus, the parallel application is paying a double penalty!

3. We want to encourage applications programmers to parallelize their codes because the future of supercomputing lies in that direction.

4. We have observed times (on the weekend and at nights) when only large memory serial jobs are in the system. This leads to CPU starvation and significantly reduced system throughput. In the short term this problem can be alleviated by restricting the maximum size of jobs, but this defeats the purpose of having supercomputers in the first place!

5. When parallel applications run, they tend to exchange less often and waste fewer cycles spin waiting if their processes run simultaneously (see Section 7).

The way that Gang scheduling is implemented in EDSEMS is that at all times there is exactly one job who is the "owner" of the system. If that job needs additional CPUs, the scheduler preempts other jobs (not processes) and takes away their CPUs. When the owner's time slice runs out, the job goes to the end of the ready queue, and a new owner is chosen.

Depending on the workload, the number of processes per job can vary widely. Typically, there are not enough processes in the owner to utilize all available CPUs (see Section 7). If the owner is the only job allowed on the CPUs at a time, then the CPU idle time will be high. Therefore, in order to increase throughput, it is necessary to schedule additional processes and jobs to the idle CPUs. However, it is important that entire jobs are scheduled if at all possible, in order to keep the level of parallel overlap high.

To solve this problem, we allow multiple jobs to be running at once, but each running job has a different priority. The owner is the job with the highest priority. If a job needs more CPUs, it is free to preempt any other running jobs with lower priority. However, if it is unable to get enough CPUs by preempting lower-priority jobs, it must release its CPUs. We call these non-owner jobs "cycle suckers." When a cycle sucker is first scheduled, it is given the lowest priority.

Unfortunately, even with cycle suckers, our simulations typically showed between 5% and 15% idle CPU time, depending on the parallelization level of the workload. This high idleness is a result of the Gang scheduling method's inflexibility in breaking up any job and running individual processes when all jobs have too many ready processes to fit on the idle CPUs. Consequently, CPUs go idle until a job can fit. Clearly a modification to Gang scheduling is needed to avert this situation.

The necessary modification comes in the form of choosing what we call "floaters" when there are idle CPUs and no jobs that will fit. In this situation, the scheduler chooses, as a last resort, a process from the job with the least number of processes, and schedule that process to the available CPU. If the owner or a cycle-sucker jobs need an additonal CPU then a floater is preferentially preempted before preempting lower-priority cycle suckers.

As a summary, when a CPU is available, the scheduler first tries to schedule it to the highest priority job that is waiting for CPUs. If there are no such jobs, the scheduler tries to schedule a job which has no more ready processes than the number of available CPUs (it also takes into consideration the jobs who are in the process of giving up their CPUs). If all jobs have too many ready processes, then the scheduler chooses a floater. The floater chosen is a ready process from the job in the system with the fewest processes.

When a cycle sucker gives up its last CPU, all cycle suckers with lower priority move up in priority. When the owner gives up its CPUs, a new owner is chosen. Since our goal is for system owners to have many processes, and jobs with many processes tend to be found at the front of the ready queue (since jobs with fewer processes are more likely to be cycle suckers), the new owner is the job at the front of the ready queue.

Even after making these changes, the system did not behave quite to our liking. In particular, in an eight-CPU system, the four-process jobs got worse treatment than the other jobs. Often, a four-process job would make its way up through the ready queue, when four CPUs would become available. Being the first job to fit, that job would be scheduled as a cycle sucker.

Typically, four or more CPUs become available only when four or more processes of the owner become blocked. In this case, it is likely that very soon the last process in the job will reach the barrier, causing the four blocked processes to become unblocked. Then, since the newly scheduled four-process job is the lowest-priority cycle sucker, that four-process job is the first to be preempted. The job then goes to the end of the ready queue without doing much work, thus increasing its ready queue latency.

For this reason, we decided to have two ready queues: a high-priority queue (HPQ) and a low-priority queue (LPQ). The idea is that jobs with many processes should typically be in the system only as owners, while jobs with few processes should only be cycle suckers. The LPQ would be for jobs with many processes and jobs entering the system, while the HPQ would be for jobs with few processes.

Jobs entering the system go at the rear of the LPQ. When a job's time slice runs out, it goes to the back of the LPQ, to give jobs with few processes in the LPQ a chance to rise to the HPQ. When a job is preempted for some other reason, it goes to the back of the HPQ. Cycle suckers are chosen by searching first through the HPQ, and then through the LPQ, for a job that will fit.

When choosing a new owner, at first thought, one might decide to choose the job at the front of the LPQ, since the jobs with many processes would seem to end up there. However, consider the following scenario in an eight-CPU system: An eight-process job is the owner. When its time slice runs out, suppose that seven of its processes are either spin waiting or blocked. This means that after the remaining process gives up its CPU, it will go to the back of the LPQ, while the other seven processes will be blocked and will go onto the blocked list (BL). Now suppose that a CPU becomes available, and there are no other jobs on the ready queues with just one ready process. Then, since our eight-process job has only one ready process, it will be scheduled. Soon that process will reach the next barrier, causing the other seven processes to unblock. However, the job will not be able to get seven more CPUs, so it will give up its CPU, causing it to go to the back of the HPQ.

The last thing we want is for an eight-process job to be on the HPQ. Since the LPQ is rarely empty, that job will seldom get a chance to be an owner again. For this reason, the owner is chosen by comparing the jobs at the front of the LPQ and the HPQ, and taking the job with the most processes, giving preference to the HPQ in case of a tie. This allows the eight-process job to leave the HPQ quickly.

4 Workload Characterization

4.1 Background

The value of various scheduling methods for shared memory MIMD parallel computers depends heavily on the workload on the computer. If the workload is totally serial (multiple jobs streams) the simple minded approach of Dog-Eat-Dog scheduling is fully justified and the expense of keeping track of both jobs and processes is not warranted. Similarly, if the workload is predominantly parallel with no synchronization between the processes of a job, it really makes no difference if they are coscheduled or not!

We are interested in the effects of scheduling on parallel scientific computation. Primarily, we wish to determine if the two competing forces (system throughput and job turnaround time vs. coscheduling of parallel applications) can both be accommodated in some fashion. Therefore, we focus our attention on a workload model based on parallel applications with frequent synchronization requirements. The synchronization requirements modeled are based on the *barrier* construct. This construct requires that all tasks (portions of parallel work) must reach a certain point (said to be the *barrier point*) in the computation before any are allowed to proceed. This construct is used extensively in the Cray Research, Inc. style multitasking parallel execution model and is based on the idea that the programmer describes the control of each task [3]. This construct is quite useful in scientific computation. For instance, in the Alternating Direction Method (ADI) for solving time dependent partial differential equations, one sweeps through a grid in one direction, giving a few grid lines to each task, and then sweeps through the grid in a perpendicular direction again giving out a few grid lines to each task. Before the computation begins one must be certain that all the needed data is available for use. Similarly, before the second sweep can begin, the first sweep must be complete. Barriers are one way to guarantee the required data integrity.

In determining how to model the barrier synchronization of processes within a job, one must know several parameters: how long does the application run; how often does synchronization occur; how well

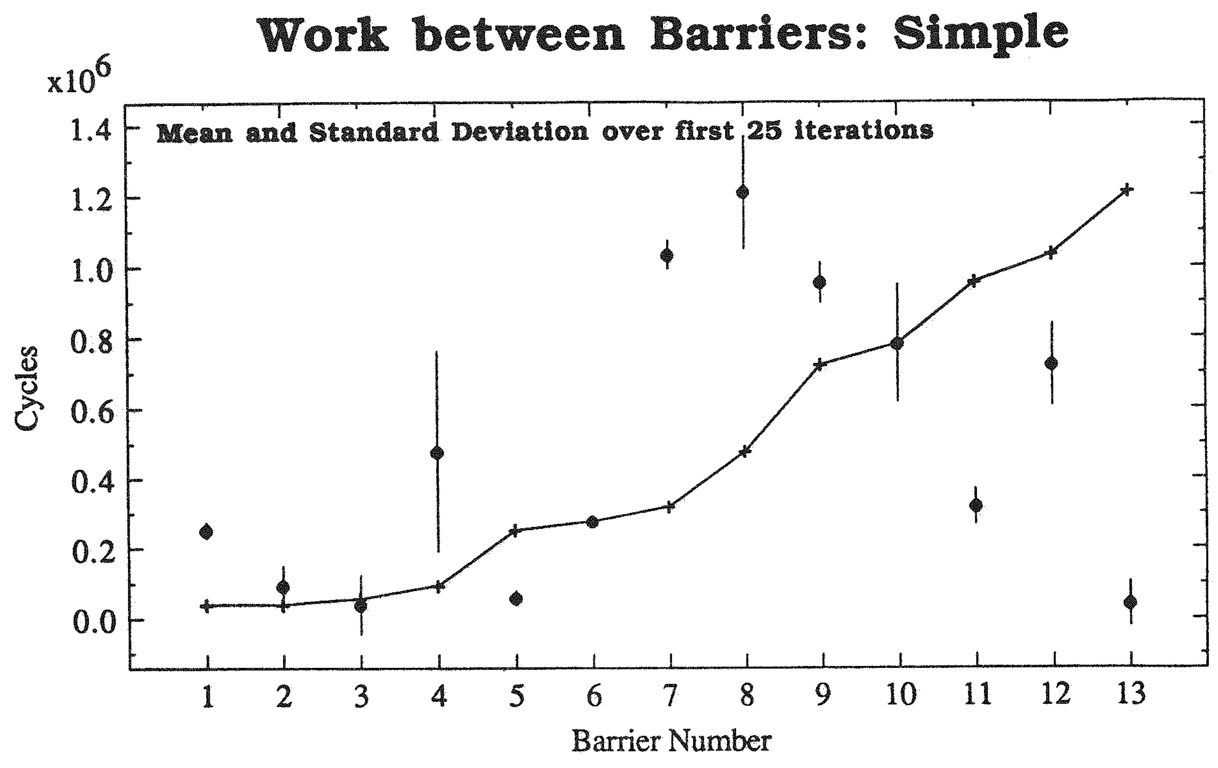


Figure 1: Work between barriers in parallel simple. The dots with error bars are the mean and plus/minus one standard deviation of the work between the 13 work barriers. The average is taken over the first 25 iterations. Sorting the work between barriers yields the curve; which looks vaguely exponential with some small amount of noise added in.

are the processes balanced (i.e. do they all do the same amount of work or do some processes wait significantly longer at barriers than others)? A detailed description of these issues is discussed in [8]. Suffice it to say that we adopted a model based on a parallelized version of the SIMPLE code [4]. The SIMPLE code solves a hydrodynamics and heat conduction problem with (2-Dimensional) cylindrical geometry. The SIMPLE execution flow starts with a single process and then immediately generates any additional processes needed for the run, sets up the initial data (in parallel) and then begins the first time step. For the parallelization of SIMPLE two locks (one for I/O and another for global dot product accumulation) and 17 or so barriers are required. The solution method SIMPLE employs timesteps initial values for temperature, pressure, density and grid positions using explicit hydrodynamics and an ADI technique for heat conduction. Each time step requires 13 barriers. The timestepping loop is repeated until the user-specified simulation termination time is reached.

A four process SIMPLE benchmark utilizing a 256×256 grid and 86 timesteps runs in about 28.3 Cray X/MP416 CPU seconds on a dedicated machine. CPU seconds is the charge rate for all four processes. The real time turnaround was much shorter. Production codes typically use much larger grids and many more timesteps and hence have prolonged runtimes. This SIMPLE benchmark result implies that each process executed $86 \times 13 = 1,118$ barriers. By looking at a tracefile of the parallel execution [9], we determined that each process does about 5.4 seconds of useful work (i.e., not counting task creation, process suspend, spin wait times and tracing overhead). Hence, the useful work between barriers is, on average, $5.4 \div 1,118 \approx 4.8$ milliseconds (see Figure 1, where the time is given in Cycles $= 8.5 \times 10^{-9}$ seconds).

We model the SIMPLE based workload with the following assumptions:

1. Jobs enter the system with the number of processes between one and the number of CPUs on the machine who participate in the computation for the duration of the job. Since SIMPLE starts as

a single process and then very quickly becomes many processes which live until the bitter end of the computation, this is a reasonable assumption and simplifies the bookkeeping enormously.

2. Job runtimes are determined by the number of barriers they perform and the average amount of work between barriers they do. This is exactly what SIMPLE benchmark does: modulo the non-trivial process creation time. This assumption implies that the cumulative CPU demand for jobs is linear in the number of processes in the job.

3. The work that each process does between specific barriers is close to, but not exactly the same as, the work the other processes do. In other words, the applications in the system are generally well load balanced. This assumption may not be valid (cf. [8]), but gives a "worst case scenario" for parallel scheduling algorithms. This is due to the fact that if the applications are well balanced, then any process wait times at barriers is known a priori to be introduced by deficiencies in coscheduling the processes of the job. Since we are interested in determining the effectiveness of scheduling parallel applications this "worst case scenario" is justified.

4. We do not model the memory requirements of a job nor worry about how many jobs fit into the "real memory" of a Cray type parallel supercompter or page faulting in "virtual memory" IBM type parallel mainframes. We feel this added complexity will only obscure the results.

5. We do not model I/O requirements of the workload. This is an area for further research: a model for parallel applications' I/O requirements is needed.

6. We do not model terminal interaction nor the operating system requirements for messages (system interrupts). Again, this is an area for further research. Although models for terminal interaction exist, models for message based operating systems do not.

To implement the above assumptions, we use the following scheme. When the job is created, the number of barriers it will have is determined by sampling a normal distribution. The mean and the standard deviation of this distribution are simulation parameters (`NBMean` and `NBStdDev`, see section 2). By using a normal distribution we can tune the workload to have a wide variety of run times (by making the standard deviation large) or make all the jobs identical (by making the standard deviation small). Also determined at job creation time is the mean work time between barriers. This mean work time is also generated from a normal distribution whose mean and standard deviation are simulation parameters (`SIMMean` and `SIMStdDev`, see section 2).

The actual CPU demand for each process between barriers is computed based on this mean. Since work between barriers can be thought of as a waiting time, an exponential distribution is sampled to get the "job-global" per-process work time before the next barrier. To model load imbalance we add some "white-noise" to the job-global per process work time to compute each process' actual work time. This noise is generated by sampling a normal distribution with mean 0. The standard deviation of this distribution is a job parameter, and that job parameter is generated at job creation by sampling yet another normal distribution whose mean and standard deviation are simulation parameters (`SISMean` and `SISStdDev`, see section 2).

The number of processes for each job is a user-specified distribution. For each value between 1 and the number of CPUs, the user specifies the probability that the job will have that many processes. This makes it possible to generate any kind of distribution for the number of processes per job.

The final workload characteristic concerns how often a new job arrives. Three methods are currently provided. The first method (the "Min" method) tries to keep the number of processes in the system constant. The number of processes in the system is a simulation parameter. In the second method (the "Load" method) tries to keep the system load constant by adding jobs when the load is less than a user specifed minimum. This method is different from the Min method because blocked processes do not contribute to the system load. In the third method (the "Exponential" method), the arrival of new jobs is determined by an exponential distribution. The mean arrival rate is a simulation parameter. Neither of these methods provides a truly accurate model of an actual system.

Let $\texttt{Exp}(\lambda^{-1})$ denote a sample from a random Exponential variable with mean λ, and let $\texttt{Normal}(\mu, \sigma)$ denote a sample from a random Normal variable with mean μ and standard deviation σ.

When a job is started, it is given the following parameters:

- Number of Processes $= P \sim$ user-specified process-per-job discrete distribution (`ParArray`)
- Number of Barriers $= B \sim$ `Normal(NBMean,NBStdDev)`
- Mean Base Work Time between barriers $= \overline{W} \sim$ `Normal(SIMMean,SIMStdDev)`
- Mean "Noise" in process synchronization intervals $= V \sim$ `Normal(SISMean,SISStdDev)`

When the job as a whole is ready to resume after a barrier, the job-global per process work time before the next barrier is $W \sim$ `Exp`$(\overline{W})$. Each process' actual CPU demand before the next barrier (the amount of work it will do) is $W +$ `Normal`$(0.0, V)$.

If the "Min" job generation method is being used, then the number of processes in the system at a time is kept nearly constant at `MinProcsInSystem` by generating jobs with P processes until at least `MinProcsInSystem` process are in the system. If the "Exponential" job generator is used, then after generating a new job, the time that elapses before generating another job is given by `Exp(DelayMean)`.

5 Implementation

5.1 Underlying Structure

Before writing the simulator, it was necessary to decide what type of simulator it should be. Specifically, we had to determine whether it should be time-based or event-driven.

In a time-based simulation, each step of the simulation represents a fixed amount of time. At each step, the modules of the simulation must communicate with each other. Each module bases its next actions on the previous actions of the other modules.

An event-driven simulation works in much the same way as a time-based simulation, except that the modules actually set "events" for the future. When the time comes for a module's event, that module performs its action, possibly stimulating other modules in the simulation.

A time-based simulation works best when state changes occur in fixed time intervals (i.e., the time scales of various aspects of the simulation are of the same order of magnatude), or when all or most of the simulation modules change state at every time step. An event-based simulation works best when events do not happen at regular intervals and few modules are affected by each event. Clearly, in a simulation like this where time scales range from micro-seconds to seconds, an event-driven simulator makes the most sense. For this reason we decided to use an event-driven simulation.

5.1.1 The Event Director

In this implementation, the "event director" runs the simulation. Any module of the simulation can set events with the director or cancel previously set events. For example, when a process gets a CPU, the scheduler will set an event to preempt that process when its time slice runs out. However, if the process gives up its CPU before its time slice runs out, then the scheduler must cancel that preemption event.

The director contains four methods:

- `SetEvent()`: This function takes as arguments the function to be called, a pointer to that function's (single) argument, and the time that the function should be called.
- `CancelEvent()`: To cancel a previously set event, `CancelEvent()` must be called with the same arguments that were passed to `SetEvent()`.
- `GetTime()`: This function simply returns the current simulated time.
- `NextEvent()`: This function tells the director to call the next chronological event. The main program, after setting some initial "seed" event to get the action started, simply calls `NextEvent()` over and over again inside a loop.

5.1.2 The Job Generator

The job generator creates new jobs. First, it generates the job parameters, which are: number of processes, number of barriers, mean work time between barriers, and standard deviation of noise in synchronization intervals. Then it initializes a job with those parameters, and enters that job into the scheduler.

The generator also determines when to generate the next job. The "Exponential" generator determines the delay before generating the next job, and sets an event for that time. The "Min" generator, however, determines at each simulation step whether another job needs to be generated in order to keep a minimum number of processes in the system. The "Load" generator, like the "Min" generator, determines at each simulation step whether another job needs to be generated. With this generator, the minimum number of processes in the system is dynamically tuned so that the user specified system load is maintained. To do this a moving average of the instantaneous system load (as opposed to the time weighted average system load) is used to measure the load. An event is generated every so often (most likely 10×`GlobalOverhead`) to compute the instantaneous system load and update the moving average. Periodically, (most likely after 100 moving average updates), a line is fit to the load moving average and an estimate of the number of processes required to keep the system load at the user specified value made.

The job generator methods, which are actually function pointers, are:

- `InitGen()`: Initializes the private data. It also makes sure that `ParArray` is valid.
- `GenNewJob()`: This simply tells the generator to generate a new job.

5.1.3 Jobs

Jobs are created and started by the job generator. A job owns its processes, and controls barrier synchronization. The job methods are:

- `InitJob()`: A job is created by calling `InitJob()`, which initializes the job's private data. The operations on a job are:
- `StartJob()`: Creates its processes, initializes them, and enters them into the scheduler.
- `HitBarrier()`: When a process reaches a barrier, it calls `HitBarrier()` to inform its owning job. The job tells the process whether to wait for the other processes, whether to continue on (if it is the last process of the job to reach that barrier), or whether to give up its CPU (if it has finished execution). When the last process of the job reaches the barrier, the job tells the other processes to stop spin waiting and continue, or to become unblocked if they are currently blocked.
- `UpdateDFG()`: When a process completes and gives up its CPU for the last time, it calls `DoneForGood()`. It then calls `UpdateDFG()` to let the job know that it is completely done. When the last process calls `UpdateDFG()`, the job updates the job statistics and deallocates the job's and processes' memory in the simulator.
- `UpdatePar()`: This is called just before a process gets or gives up a CPU. `UpdatePar()` simply updates the parallel overlap statistics.

5.1.4 Processes

Processes are created and started by their owning jobs. When a process is running, it owns a CPU. Each process contains several flags which, when examined, tell the state of the process, such as: running, ready, blocked, spin waiting, or giving up its CPU. The following methods operate on processes:

- `ProcUpdateStats()`: When a process' run state changes (i.e. running, ready, spin waiting, blocked), it calls `ProcUpdateStats()`. These statistics are added to the system-wide statistics when the job completes.

- **StartProc()**: This just initializes the process' private data. After calling **StartProc()**, the owning job takes care of entering the process into the scheduler.
- **GotCPU()**: When the scheduler assigns a CPU to the process, **GotCPU()** is called. **GotCPU()** takes the CPU and starts doing work.
- **GiveUpCPU()**: When the scheduler preempts a CPU, that CPU calls **GiveUpCPU()** to tell the process to start giving up its CPU. The process adjusts its remaining time slice and starts the context switching.
- **UnblockSync()**: When the last process of the job calls **HitBarrier()**, the job calls **UnblockSync()** on all of its processes. If a process is spin waiting, it starts its next segment. Otherwise, the process becomes unblocked and starts competing for a CPU again.
- **ClearCPU()**: When a process finishes its context switching overhead, it tells its CPU that it is finished. The CPU then calls **ClearCPU()** to tell the process officially that it no longer has a CPU.
- **ProcWaiting()**: This function returns whether or not the process is ready. Since **ProcWaiting()** is called so much, it was made into a macro, rather than a function.
- **GetTimeSlice()**: This function returns the process' remaining time slice.

5.1.5 Process Queue

The process queue methods allow inserting and deleting a process anywhere in the queue. Standard queue and stack functions are provided as well. The process queue methods are:

- **InitQueue()**: This function is called when a queue is first created. It simply sets the head and tail pointers to **NULL**.
- **PutFront()**, **PutRear()**, **GetFront()**, **GetRear()**: These are the standard queue/stack functions. They allow pushing and popping a process to/from the front or rear of the queue.
- **PQSize()**: This function returns the number of elements in the queue.
- **DeleteElt()**: This function allows a specific process to be deleted from the queue. If the process was not in the queue, **NULL** is returned; otherwise the process is returned.
- **InsAfter()**: This function allows a process to be inserted in a specific location in the queue. Unfortunately, if **InsAfter()** is called, the caller must have knowledge of the queue structure, since a pointer to a queue element must be passed. This pointer is the element after which the process is to be inserted.
- **PQPrint()**: Although **PQPrint()** is not called in the simulation, it provides a method for printing the contents of a queue. It is very useful to call **PQPrint()** while stepping through the simulation with dbx or dbxtool, especially when designing a new scheduler.

5.1.6 The CPUs

The CPUs are controlled by the scheduler. When they are not idle, they are "owned" by a process. The following methods operate on CPUs:

- **InitCPU()**: When the scheduler creates the CPUs, it calls **InitCPU()** on each one, in order to initialize each CPUs private data.
- **CPURelease()**: When a process gives up its CPU and finishes the context switching overhead, it calls **CPURelease()**. The CPU then resets its owner field, tells its owning process to reset its CPU field and finally lets the scheduler know that it has just become idle. Note that this is only called when the process still has work left to do (see **DoneForGood()**).

- `DoneForGood()`: This is just like `CPURelease()`, except that `DoneForGood()` is called after the process is completely finished and is releasing the CPU for the last time.
- `CPUAvailable()`: This function simply returns whether the CPU is idle.
- `CPUAssign()`: When the scheduler assigns a CPU to a process, it calls `CPUAssign()`. `CPUAssign()` sets its owner field, and then tells its owner to start working.
- `CPUPreempt()`: When a process' time slice runs out, or the scheduler preempts the process for any other reason, it calls `CPUPreempt()` on that process' CPU. The CPU then tells the process to start giving up its CPU. The scheduler and the CPU also pass a status variable to inform the process why it is being preempted.
- `AvailableCPUs()`: This function simply returns a count on the number of idle CPUs in the system.

5.1.7 The Scheduler

While the director is the heart of the simulator, the scheduler is the most important part. There are several methods, which have different complexities in different schedulers. All of these methods are accessed by calling function pointers, which are initialized to the methods corresponding to the chosen scheduler at the beginning of execution. The methods are:

- `InitSched()`: This function creates and initializes the CPUs and the ready queues and blocked lists.
- `SchEnter()`: When a process is created, it calls `SchEnter()` to register itself with the scheduler. The scheduler typically puts the process on the ready queue, and it might try to assign a CPU to the process.
- `TimeSliceRanOut()`: A process calls `TimeSliceRanOut()` when it is about to give up its CPU after being preempted because its time slice ran out. This function restores its time slice to its original value, calls `CPURelease()` and usually does something with the process, such as put it on the ready queue.
- `ProcBlocked()`: This is just like `TimeSliceRanOut()`, except that a process calls `ProcBlocked()` when it gives up the CPU because it blocked. The scheduler might put the process on a blocked list instead of a ready queue.
- `ProcDone()`: A process calls `ProcDone()` when it is about to give up its CPU because the process completed. `ProcDone()` is usually identical to `TimeSliceRanOut()`, except that the scheduler probably will not put the process back on the ready queue.
- `SchProcPreempted()`: This is just like `TimeSliceRanOut()`, except that a process calls `SchProcPreempted()` when it gives up its CPU because the scheduler preempted it for some reason besides its time slice running out. In the current schedulers, `TimeSliceRanOut()` and `SchProcPreempted()` behave identically, but in other types of schedulers that might be implemented, they might behave differently.
- `IdleCPU()`: When a CPU becomes idle, it calls `IdleCPU()`. The scheduler will usually cancel any preemption event that was set for that CPU, and it might try to assign that CPU to the next ready process.
- `BecameUnblocked()`: When a process changes from being blocked to being unblocked, it calls `BecameUnblocked()`. This function will probably move the process from a blocked list to a ready queue.
- `JobUnblocked()`: When the last process in a job reaches a barrier, the job reactivates the spin waiting and blocked processes in that job. After reactivating them, the job calls `JobUnblocked()` to let the scheduler know that all the processes have been reactivated. Usually, the scheduler will then try to assign available CPUs to ready processes.

5.1.8 Statistics

There are three statistical gathering methods: job statistics, queue size/load average statistics, and CPU utilization statistics. Each of these has four methods (although these are not the actual method names):

- `Init()`: Initializes the data, and possibly allocates memory for some of the data.
- `Restart()`: Re-initializes the data, but without allocating any additional memory.
- `Update()`: This is usually called just before some change is made, so that the statistics can be updated.
- `Print()`: This prints out the statistics gathered so far.

5.1.9 Histograms

The histograms are similar in operation to the statistical objects, except for the way they are printed. There are three methods for histograms:

- `InitHist()`: This is called to initialize the histogram. The arguments passed to `InitHist()` are the minimum and maximum values and the number of "buckets".
- `HistData()`: When a value is passed to `HistData()`, the function decides which bucket the value goes in and increments the number of values in that bucket.
- `HistPrint()`: This function graphically displays the histogram. It also shows the percentage of values that fell in each bucket.

5.2 The Life of a Process

Here are the major steps a process goes through from creation to completion:

- The generator creates a new job and calls `StartJob()`.
- `StartJob()` creates a process, initializes it with `StartProc()`, and enters it into the scheduler with `SchEnter()`.
- When the process gets scheduled to an available CPU, the scheduler calls `CPUAssign()`. The CPU calls `GotCPU()` to tell the process to start working.
- If the process' time slice runs out while it is still working, or the scheduler wants the process to give up its CPU for some other reason, then the scheduler calls `CPUPreempt()` on the CPU. `CPUPreempt()` then calls `GiveUpCPU()` on the process, and the process starts giving up its CPU.
- When the process reaches a barrier, it calls `HitBarrier()` and starts spin waiting. When all the processes in the job have reached the barrier, the job calls `UnblockSync()` on all the processes and then calls `JobUnblocked()`. If the process is spin waiting when `UnblockSync()` is called, then it just starts doing work on the next barrier. If the process is blocked when `UnblockSync()` is called, then it calls `BecameUnblocked()` to let the scheduler know that it is now unblocked.

  When the process starts spin waiting, there is a maximum time that it will spin wait. If it spin waits for that amount of time without `UnblockSync()` being called, it will start to give up its CPU.
- When a process finishes giving up its CPU, it calls either `TimeSliceRanOut()`, `ProcBlocked()`, or `SchProcPreempted()` to tell the scheduler that the CPU is now available and to let the scheduler know why it gave up its CPU.

- When the process reaches its last barrier and calls `HitBarrier()`, the job lets the process know that it has completed. Then the process starts giving up its CPU. When it finishes giving up its CPU, it calls `DoneForGood()` to let the scheduler know that its CPU is now available and that the process is done. Then the process lets its owning job know that it is completely done by calling `UpdateDFG()`. When the last process in the job calls `UpdateDFG()`, the job updates the final job statistics and deallocates memory for the job and its processes.

5.3 Data Structures

Following are the important data structures of each component:

5.3.1 Job Data Structure

As defined in file `job.h`, the job fields are:

- `NumProc`: The number of processes in the job. Initialized at job creation.
- `Proc`: An array of process pointers. `Proc` is dynamically allocated at job initialization, with size `NumProc`.
- `StartTime`: The starting time of the job. `StartTime` is used only to calculate job turnaround time.
- `ParLastUpdate`, `ParRunTime`, `ParSum`: These fields are used to calculate the parallel overlap.
- `LeftDFG`: This is initialized to `NumProc` and is decremented each time a process completes and calls `UpdateDFG()`. When `LeftDFG` reaches 0, the job knows that all processes in the job have completed.
- `NumBar`: The total number of barriers in the job. It is initialized when the job is created.
- `Mean`: The mean of the mean work between barriers. It is initialized when the job is created.
- `StdDev`: The standard deviation of the "noise" in work between barriers. It is initialized when the job is created.
- `Reached`: The number of barriers the job has reached. When `Reached` + 1 == `NumBar` and a process calls `HitBarrier()`, then the process has reached its last barrier and can give up its CPU for good.
- `ProcsLeft`: The number of processes that have not yet reached the current barrier. When a process calls `HitBarrier()`, `ProcsLeft` is decremented. When `ProcsLeft` gets to 0, the job reactivates spin waiting processes and unblocks blocked processes. At the beginning of each synchronization interval, `ProcsLeft` is initialized to `NumProc`.
- `FamPriority`: In Family scheduling, `FamPriority` is used to tell the priority of the job.
- `ThisBarrier`: This is the mean base work time for the current barrier.
- `Ident`: This is the unique name of the job. It is used only to distinguish the jobs when examining debugging output.

5.3.2 Process Data Structure

As defined in file `process.h`, the process fields are:

- `Owner`: This is a pointer to the job to which the process belongs.
- `myCPU`: If there is a CPU assigned to the process, then `myCPU` is a pointer to that CPU. Otherwise, this field is set to `NULL`.

- `WhichList`: If the process is in a ready queue or blocked list, `WhichList` is a pointer to that list.
- `RemainingTime`: This is the remaining time that the process must run before it reaches the next barrier.
- `StartTime`: This is the time that the process got its CPU.
- `FinishTime`: If the process is running, `FinishTime` is the time when it will reach its next barrier. If the process is spin waiting, `FinishTime` is the time when, if it is not reactivated, it will stop spin waiting and begin giving up its CPU.
- `SpinWaitDelay`: This is the maximum time that the process will hold on to its CPU and wait for reactivation after it reaches a barrier.
- `Overhead`: This is the amount of time it takes for the process to release its CPU.
- `TimeSlice`: This is the remaining time slice of the process.
- `InitialTimeSlice`: When a process' time slice runs out, it is preempted and its `TimeSlice` is reset to `InitialTimeSlice`.
- `Ident`: This is the unique name of the process. It is used only to distinguish the processes when examining debugging output.
- `PrevTime`, `WorkType`, `WorkTimes`: These fields are used to keep track of certain statistics, such as: time spent running, ready, spin waiting, blocked, and doing system overhead; turnaround time; and number of exchanges.

In addition, each process has the following flags, which can be used to determine its run state:

- `DoingOverhead`: This is set when the process is giving up its CPU.
- `SpinWaiting`: This is set when the process is spin waiting, waiting to be reactivated.
- `BlockedSync`: This is set when the process is blocked. It is set as soon as the process starts giving up its CPU to become blocked.
- `Inactive`: This is set when there is some reason that the process should not be scheduled. For example, after a process completes, but before all the processes in the job have completed, `Inactive` is set, so that the scheduler will know not to try to schedule it. Currently, `Inactive` is used only in the Gang scheduler.

6 Interpreting the Simulator Output

After reading in the input files, the value of each input parameter is displayed. In addition, if the random number generator was seeded, then that seed is displayed. When the generator is seeded randomly (`RandomSeed` = 0), printing the seed makes it possible for the user to recreate the simulation.

Other than the displaying of the input parameters, the outputs from batch mode and interactive mode are completely different. Interpreting the output in each mode is described below.

6.1 Batch Mode

In reading this section one might want to have Appendix A close at hand for easy reference. The frequency of output in batch mode is determined by the input parameter `OutputDelta`. Short-term and long-term data are displayed every `OutputDelta` units of simulated time. At the beginning of every output cycle the date and time are given. Next the current simulation time (`Current Time`, in micro-seconds), amount of CPU time used in the simulation so far (`Cpu Time`, in micro-seconds) the ratio of simulation time to CPU time (`Ratio`) and the moving average instantaneous load (`Load`, see Section 5.1.2 for more information on this quantity). Next the period simulation over which the following statistics were gathered is printed. The above information makes it easy to determine the length of time the simulation has run and how long it took to compute the system. This is useful in determining how much longer the simulation will run (when looking at intermediate results) and how much CPU time it will require.

Following the header information are the short-term and/or long-term statistics. Both are given as groups of tables. The data in each table has a unique mnemonic identifier at the beginning of the line to make filtering out specific tables with **grep** easy. For every table the first two characters are `ST` or `LT` to denote short-term or long-term statistics. The next one, two or three characters denote the information in the table:

Mnemonic	Description
`TP`	Throughput
`APP`	Average Per-Processes
`SDS`	Standard Deviation Per-Process
`Q`	Queue Length
`CPU`	CPU Utilization

Short-term statistics, which are collected between the previous output time and the current time, are displayed first if they are displayed at all. When the user chooses `OutputDelta` to be the same as `SimLength`, then the short-term statistics are identical to the long-term statistics and are therefore not printed. The job and process completion (throughput) statistics are printed first. These statistics are grouped so that jobs with the same number of processes in them are reported on a single line. Then overall system totals are given. This lets the user know how many jobs completed with each level of parallelism and what the overall system performance was. For example, the line with "`ProcsInJob` = 2", "JobsComp = 60" and "`ProcsComp` = 120" means that for this short-term statistics gathering period (from simulation time 900000954.300896 to time 1000000009.050145) 60 jobs with 2 processes in them completed. This added 120 processes to the completion total.

Next, for each level of parallelism, **per-process** average and standard deviation statistics are displayed. That is, the first table is the per-process mean (the second table is the per-process standard deviation) averaged overall jobs with the same level of parallelism (i.e., `Proc` = 8 statistics are computed over all processes belonging to completed jobs with eight processes in them). The final line is the system wide (labeled `Tot`) average and is computed over all processes in the system. All time units are in mili-seconds: apposed to micro-seconds like the input. The statistics displayed in the columns are:

- `ReadyQu`: ready queue latency. The time each process spent while it was ready to run but not scheduled on a CPU.
- `UserWrk`: user work time. The amount of time each process spent doing useful user work. On average, this value should be `SIMMean` × `NBMean`.
- `Sys`: system work time. The amount of time each process spent context switching.
- `SpinW`: spin wait time. The amount of time each process spent waiting in a CPU for other processes in the job to synchronize.
- `BlockQu`: blocked queue latency. The amount of time each process spent blocked due to interprocess synchronization.
- `Ovrlp`: average parallel overlap. For each job, the time weighted average number of processes with CPUs, when at least one process in the job has a CPU. This is a measure of how well the scheduling method is delivering the machine to individual jobs. For n-process jobs, the mean is between 1 and n, with n being optimal. For 1-process jobs, the mean is 1 and the standard deviation 0, no matter what the scheduling algorithm.
- `Turn Around`: average turnaround time. The amount of time the processes of a job (and hence the job itself) spends in the system, from entry to completion.

- `Cntx Swtch`: average number context switches. The number of context switches each process required averaged over processes that completed.

The next short-term statistics displayed are the average lengths for the various queues and the system load average. These statistics are displayed in two columns, the first for the mean and the second for the standard deviation. The system load at a given machine state, ρ_i, is defined as the number of ready processes plus the number of running processes, divided by the number of CPUs. The system load average, $\overline{\rho}$, is the time weighted average of the system load, viz.:

$$\overline{\rho} = \frac{\sum_{i=1}^{NStates} \delta T_i \times \rho_i}{\sum_{i=1}^{NStates} \delta T_i}$$

Where $NStates$ is the number of states the machine has been in during the statistics gathering period, δT_i is the length of time the machine was in state i and ρ_i is the system load at state i.

The final short-term statistics displayed are the CPU utilization statistics. There are two columns: the first column represents the time spent in that area, and the second column represents the percentage of total CPU time spent in that area. The second column should sum up to 100%. The breakdown of CPU utilization is:

- `Idle Time`: percent of total CPU resources spent idling. To be specific Idle Time is computed via

  $$100 \times \frac{\sum_{i=1}^{NCPU} Idle_i}{\texttt{OutputDelta} \times NCPU},$$

  where $Idle_i$ is the amount of time processor i was idle.

- `User Work Time`: percent of total CPU resources spent running user code, exclusive of spin wait time. This is computed in a similar fashion to Idle Time, viz.:

  $$100 \times \frac{\sum_{i=1}^{NCPU} User_i}{\texttt{OutputDelta} \times NCPU},$$

  where $User_i$ is the amount of time processor i was running the user code, but not spin waiting.

- `System Overhead`: percent of the total CPU resources spent process context switching. This statistic does not include spin wait time.

- `Spin Wait Time`: percent of the total CPU resources given to processes that were actively spin waiting.

The next group of data displayed in the output stream is the long-term statistics. The format of the long-term statistics is identical to the short-term statistics, but long-term statistics are collected from the beginning of the simulation to the present time and they are never reset.

Finally, the long-term histograms are displayed. The minimum and maximum values of each bucket are displayed at the left of each row. In the middle of each row appears a number of asterisks, corresponding to the number of items in that bucket. At the right of each row, the percentage of total items in that bucket is displayed. The top bucket contains items that were greater than the maximum value for the histogram, while the bottom bucket contains items that were less than the minimum value (see `InitHist()`).

The rows of numbers at the bottom of the histogram are meant to be read vertically, from top to bottom (see Figure 2). These are meant to be a guide to how many items are in each bucket. For example, in the piece of a histogram pictured below, the bucket shown contains about 110 items, since the last asterisk is in the column that contains the digits "1" "1" "0" (110). This bucket contains 11.19% of the total items.

The following long-term histograms are displayed:

- `Processes per job`: This histogram shows how many processes each job which was started had. The percentages shown at the right of the rows should compare closely with the values of the user input array `ParArray`. If there is a discrepancy, then the `SimLength` is too short and all the other statistics are subject to doubt.

```
|
|***********                                       11.19%
+--------------------------------------------------
00000000001111111111222222222233333333334444444444
01234567890123456789012345678901234567890123456789
00000000000000000000000000000000000000000000000000
```

Figure 2: Portion of histogram output showing labels and a single column of asterisks.

- `Percentage of spin wait used`: When a process finishes spin waiting, the amount of time it spent spin waiting is recorded as a percentage of the maximum time it could have spent, and this value is put into the histogram. Of course, the less spin wait used the better.
- `Percentage of time slice used`: This works much the same as the Percentage of spin wait used histogram, except that it measures the amount of a process' time slice it used after it gives up its CPU. If the process completes during its time slice, and gives up its CPU for good, then an item is put into the first bucket (the one for 100% and up), so as not to bias the other buckets. If processes consistently give up the CPU before their time slice is up this indicates some combination of the following system imbalances: the distribution of work between individual processes within a job is not balanced (i.e., the parallel workload is imbalanced); the spin wait time is too short; the scheduling algorithm is not doing a good job at giving overlap to the jobs. These effects can be tuned with simulator parameters and hence one can determine the dominate cause of the problem.

The following two histograms are only displayed when the Gang scheduler is used.

- `Size of system owners`: When a new owner is chosen under Gang scheduling, the number of processes it contains is recorded in this histogram. Recall that owners with many processes are desirable.
- `Size of cycle suckers`: When a cycle sucker is chosen in Gang scheduling, the number of processes it contains is recorded in this histogram. In general, it is desirable to have cycle suckers have fewer processes in the job than owners.

6.2 Interactive Mode

```
SimTime:264671     ProcInSys:24    JobInSys:9    ProcComp:1     JobComp:1
Family    Idle:  0.33% User: 94.72% System:  0.75% SpinWait:  4.21%
CPU |0  |1  |3  |5  |3  |2  |0  |7  |
    |0  |4  |0  |0  |1  |0  |1  |0  |

LPQ |9  |8  |8  |8  |6  |   |   |   |   |   |   |   |   |   |   |   |
    |0  |5  |6  |7  |0  |   |   |   |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
HPQ |1  |1  |1  |1  |1  |   |   |   |   |   |   |   |   |   |   |   |
    |0  |1  |2  |3  |5  |   |   |   |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
BL  |8  |8  |8  |8  |8  |6  |   |   |   |   |   |   |   |   |   |   |
    |3  |4  |0  |1  |2  |1  |   |   |   |   |   |   |   |   |   |   |
```

Figure 3: Example of interactive screen displaying overall system statistics and queue status.

In interactive mode, the contents of the queues are displayed using the Curses [1] screen manipulation package (see Figure 3). In particular, the identification of each active process in the system is displayed.

The identification of a process consists of the identification of its owning job and its number within the job.

Each queue (including the run or CPU queue) is allocated one or more pairs of rows. Several processes are displayed on each pair of rows. The first row of the pair contains the job identification, and immediately below it on the second row is the process number. The contents of the queues are displayed in this manner, where the process at the front of the queue is displayed on the first pair of rows on the left.

At the top of the screen, general system statistics (gathered from the beginning of the run) can be viewed. On the first line of the terminal the simulated time (SimTime; assumed to be in microseconds), the number of processes currently in the system (ProcInSys), the number of jobs currently in the system (JobInSys), the number of processes which have completed (ProcComp) and the number of jobs which have completed (JobComp) are displayed. On the second line is the scheduling method ("DogEatDog", "Family", or "Gang"), and the cumulative CPU utilization statistics (idle time, user work time, system overhead, and spin wait time) are displayed.

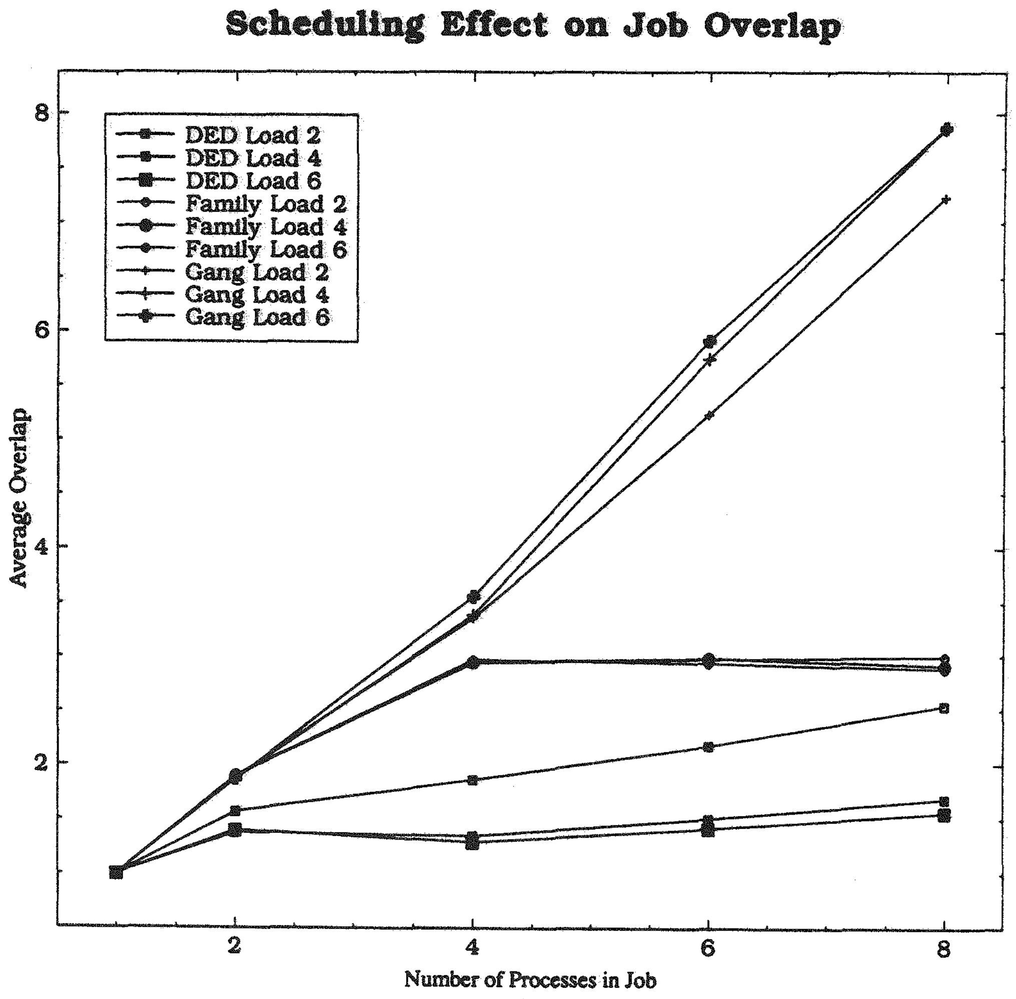


Figure 4: Parallelism delivered (overlap) to jobs with various numbers of processes in them. Job mix is half serial and half parallel.

In Figure 3, 264671 simulated microseconds have passed. There are 9 jobs and 24 processes currently in the system, and one job and one process have completed. We are using the Family scheduling method. The CPU activity is broken down as follows: 0.33% idle, 94.72% user work, 0.75% context switching and spin waiting 4.21% of the time. Processes 0.0, 1.4, 3.0, 5.0, 3.1, 2.0, 0.1 and 7.0 all have CPUs, where we are using "job index"."process index" notation. The LPQ (Low-Priority Queue) is holding processes

9.0, 8.5, 8.6, 8.7 and 6.0. In the HPQ (High-Priority Queue) we find processes 1.0, 1.1, 1.2, 1.3, and 1.5. Processes 8.3, 8.4, 8.0, 8.1, 8.2 and 6.1 are on the blocked list. It is iteresting to note that, in this example, more processes are in the blocked queue, and hence can not be scheduled, than in either the LPQ or HPQ. With Family scheduling this is often the case.

Highlighting is used to display two types of processes. If the `-s` command line flag is used, then processes which are blocked, spin waiting, or giving up their CPUs, are highlighted (displayed in standout mode). If the `-w JID` flag is used to specify several jobs to watch (and the `-s` flag is not used) then only the selected jobs are highlighted no matter what their state is.

7 Results

Simulating parallel applications on supercomputer production systems is quite difficult because most MIMD supercomputers are currently being run in multiple stream mode (i.e., many serial jobs run contemporaneously instead of parallel applications). Hence, it is impossible to predict what levels of parallelism (number of processors in a job) and efficiency (e.g., load balancing and choice of parallel algorithm) future workloads will display. However, we can look at scheduling methods from the perspective of how they allocate resources to any type of load presented to them and determine where the weaknesses of each are. This will allow us to start determining the extent of research necessary to truly understand the issues of parallel application scheduling on MIMD supercomputers.

7.1 A Moderately Parallel Workload

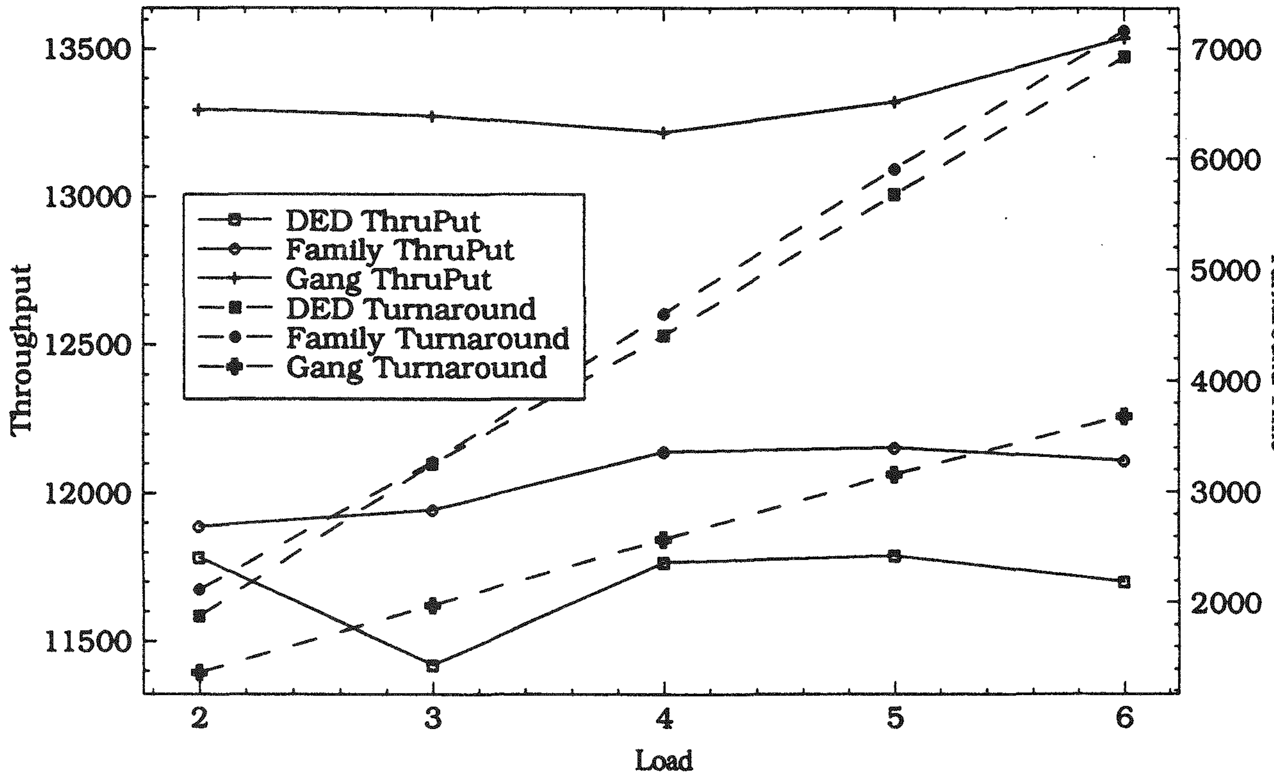


Figure 5: Throughput is larger and average turnaround time is smaller for Gang scheduling vs Dog-Eat-Dog or Family. Job mix was half serial and half parallel.

The first thing to consider is how do current and proposed scheduling methods fare with a fixed moderately parallel workload? Do the old scheduling methods deliver the machine to the parallel jobs in an

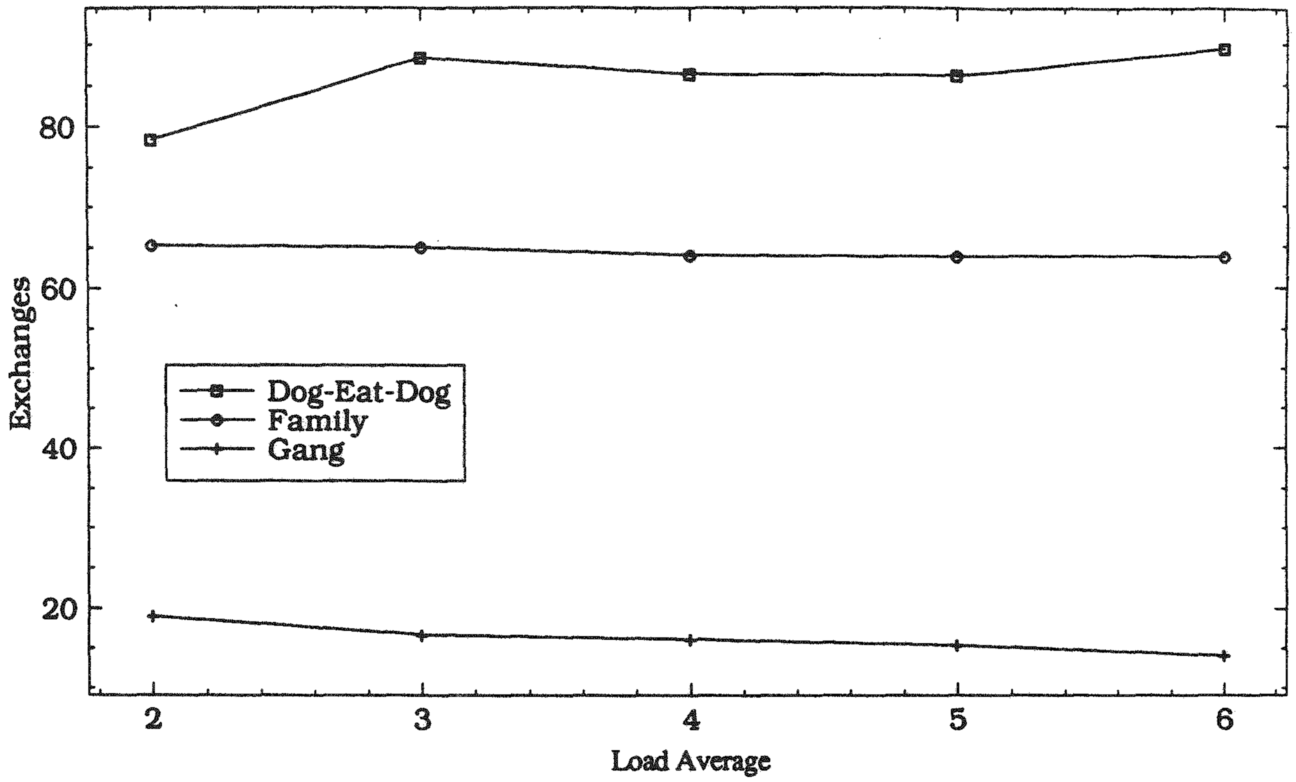


Figure 6: Number of exchanges per process (a measure of thrashing) is smaller for Gang scheduling vs Dog-Eat-Dog or Family. Job mix is half serial and half parallel.

efficient manner? In giving the CPUs to the parallel jobs with Gang scheduling, do we have to pay a penalty in throughput and/or turnaround time?

In all of the following tests we will keep most of the job and system parameters fixed and look at the effects of modifying one parameter. Unless otherwise stated the jobs and system parameters that are used in the simulations are: **SIMMean** = 4090.0, **SIMStdDev**= 409.0, **SISStdDev**= 204.5, **NBMean** = 122, **NBStdDev** = 90, **NumCPUs** = 8, **GlobalTimeSlice** = 1.0E5, **GlobalSpinWaitDelay** = 1.0E3, **GlobalOverhead** = 350.0 and **SimLength** = 1.0E9.

All of these machine and system parameters are taken from NLTSS running on a Cray Y-MP832 (6.0 Nano-Second clock cycle). Only one type of application is modeled: SIMPLE (see Section 4). The number of barriers is chosen so that the expected runtime of each job (ignoring synchronization delays, spin waiting and process exchange overhead) is about 0.5 seconds.

In Figures 4 through 6 the workload is broken up into half serial jobs (number of process per job is 1 for half of the jobs generated) and the other half of the workload is parallel (number of processes per job greater than 1 for half of the jobs generated). In particular, **ParArray**= (0.5, 0.125, 0.0, 0.125, 0.0, 0.125, 0.0, 0.125). Hence, the probability of getting a job with 2, 4, 6 or 8 processors in it is $\frac{1}{8}^{th}$ and no jobs with 3, 5 or 7 processes are considered. For this study, we consider the effectiveness of the three scheduling methods when the load is increased. Runs were made with a load of 2.0, 4.0 and 6.0. Figure 4 displays how the three scheduling methods give the requested CPUs to the jobs. From this figure we can see that Dog-Eat-Dog (DED, in the legend) never gives an overlap of more than 2.5 and as the load increases, that peak drops to about 1.5. Family scheduling does a bit better with a peak of about 3.0 and does not seem to be too affected by the change in load. Gang, on the other hand, is basically linear in the number of processes in the job: delivering over 98% of the processors requested, no matter the load. In fact, Gang scheduling gets slightly better as the load increases. This is due to the fact that when the load is higher the probability of picking the larger jobs (higher number of processes) as "floaters" decreases. So Gang scheduling is able to give the resources that jobs request, but at what penalty? We could get the same overlap behavior from dedicating the machine to one job in turn. Figure 5 shows that throughput and response time are not degraded by going to Gang scheduling over Family or Dog-Eat-Dog. In fact, we see that throughput is enhanced by around 11% because the parallel jobs do less spin waiting with Gang scheduling. Turnaround times are reduced and the reduction grows as the number of processes in the job increases. This is again due to the fact that parallel jobs spend much less time spin waiting and have far fewer exchanges to the system. The latter is observed by viewing Figure 6.

7.2 More Parallelism Anyone?

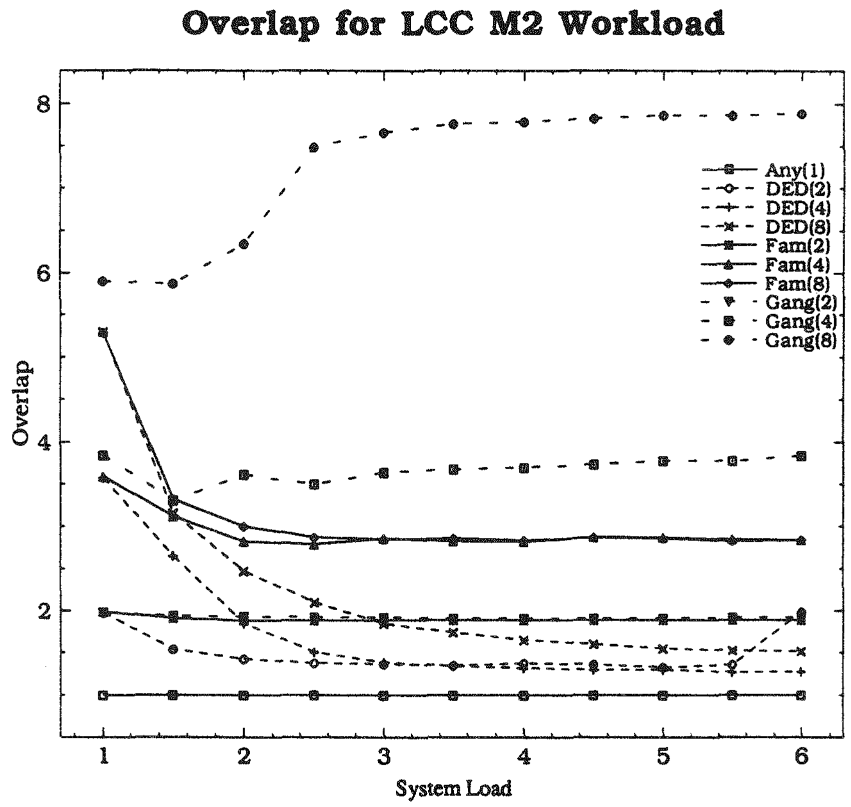
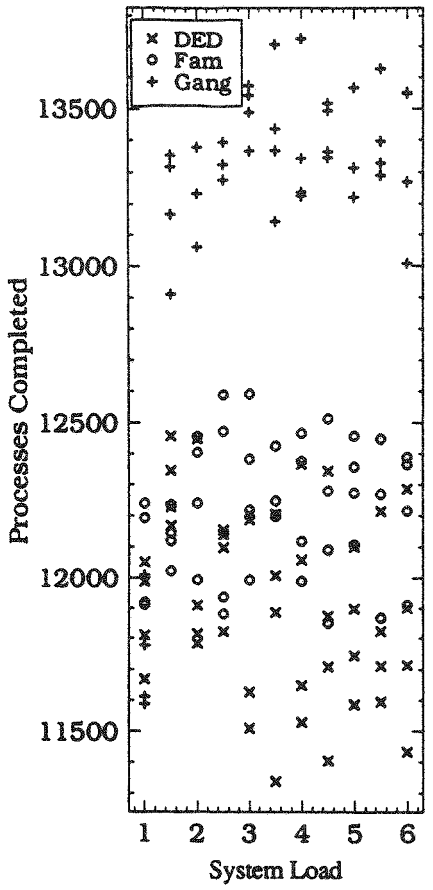


Figure 7: Low levels of workload parallelism and light load combine to cause all scheduling methods difficulty. As the load increases, so does Gang's ability to deliver overlap. On the other hand, Family and Dog-Eat-Dogs' overlap performance degrade as load increases.

Now that an understanding of the various scheduling methods with a moderately parallel workload has been established, a more complicated test can be constructed. In this test, all system parameters are held constant except for the parallelism the workload displays and the system load. Four workloads (M1, M2, M3 and M4) that are probable parallel extensions of the current serial workload at the Livermore Computing Center (LCC) are constructed (see Table 1). As we move from workload M1, to M2 through M4 the amount of parallelism is increased from most jobs being serial (80% of the jobs generated have one process) to only a few serial jobs (20% of the jobs generated have one process).

Overlap results for workload M2 are displayed in Figure 7. Similar overlap results are obtained with the other workloads. All methods deliver 1.0 overlap to serial jobs (Any(1) in the legend of Figure 7). For the level of parallelism displayed in the M2 workload it is possible to have 4 and 8 processor jobs running at the same time (with no others) at load levels 1.0 and 1.5. Similarly, we could have 2 and 8 processor jobs or 1 and 8 processor jobs running at the same time. Hence, some of the time (when the 1, 2 or 4 process job owns the machine) the 8 process job will have to be broken up. This causes the dip in the observed overlap for Gang(8) at low system load. Even with this degraded overlap, both



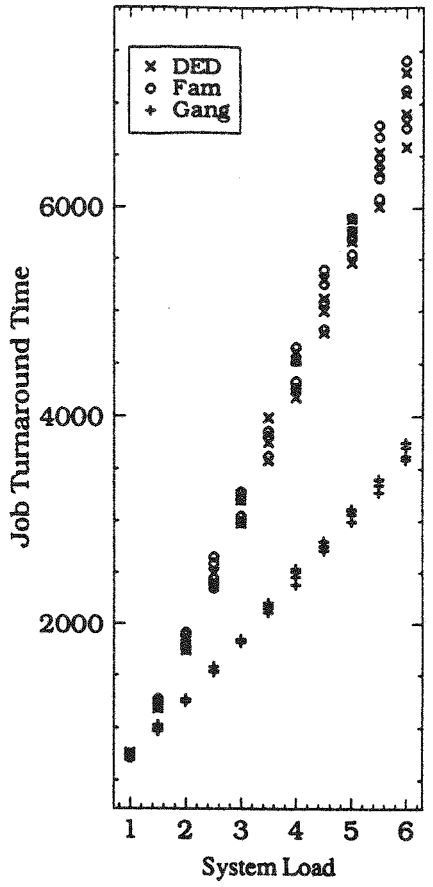


Figure 8: System throughput (on left) shows a distinct difference between Gang scheduling and the others for virtually all system loads and all workloads. Turnaround times (on right) start out the same, but increase much more rapidly for Family and Dog-Eat-Dog scheduling as load is increased for all workloads.

Dog-Eat-Dog and Family scheduling do worse than Gang for light system loads. Furthermore, their performance decreases as the load increases.

Figure 8 displays the combined results for all scheduling methods and workloads. The left hand graph in Figure 8 shows that Gang scheduling does poorly with regard to system throughput if the large parallel jobs have to be broken up. This serves as an indication of the lower limit of Gang scheduling's usefulness in lightly loaded systems. Turnaround times, on the other hand, are much less sensitive at lower system loads and hence the difference between the scheduling methods is negligible. As the system load increases, Family and Dog-Eat-Dog scheduling are much less able to respond to the load due to the higher levels of context switching. Hence, the slope of the lines in the right hand graph in Figure 8 is much larger for Family and Dog-Eat-Dog when compared with Gang scheduling.

Probability of generating the next job with specified number of processes								
	Number of Processes							
Workload	1	2	3	4	5	6	7	8
M1	0.800	0.000	0.000	0.100	0.000	0.000	0.000	0.100
M2	0.600	0.133	0.000	0.133	0.000	0.000	0.000	0.134
M3	0.400	0.200	0.000	0.200	0.000	0.000	0.000	0.200
M4	0.200	0.266	0.000	0.266	0.000	0.000	0.000	0.268

Table 1: Probability of generating a job with 1, 2, 4, or 8 processes in it for the LCC jobs classes M1, M2, M3 and M4. Jobs with 3, 5, 6 or 7 processes are not generated.

8 Conclusions

We have presented a methodology for testing scheduling methods for parallel supercomputers. The EDSEMS simulator described here is a flexible tool for evaluating various workload, machine and scheduling method models. This tool is sorely needed at present due to the lack of literature on parallel scheduling trade-offs.

Preliminary results with the EDSEMS simulator indicate that Gang scheduling is far superior to Dog-Eat-Dog and Family scheduling for shared memory supercomputers with various parallel workloads when synchronization is considered. It is the real effect of processes spin waiting and giving up CPUs at barrier synchronization points (a control based synchronization which requires that all processes in a computation reach a certain point before any are allowed to proceed) that introduces enough system overhead to warrant special treatment of parallel jobs from a system throughput and turnaround time perspective.

References

[1] Kenneth C. R. C. Arnold. *Screen Updating and Cursor Movement Optimization: A Library Package.* Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720. In UNIX Programmer's Manual, Supplementary Documents.

[2] Cray Research, Inc., Distribution Center, 2360 Pilot Knob Road, Mendota Heights, MN 55120. *Autotasking User's Guide.*

[3] Cray Research, Inc., Distribution Center, 2360 Pilot Knob Road, Mendota Heights, MN 55120. *Cray Y-MP and Cray X-MP Multitasking Programmer's Manual.*

[4] W.P. Crowley, C.P. Hendrickson, and T.E. Rudy. The simple code. Technical Report UCID-17715, University of California, Lawrence Livermore National Laboratory, PO Box 808, Livermore, CA 94550, 1979.

[5] Jed Donnelley. Components of a network operating system. *Computer Networks*, 3:389–399, 1979.

[6] Leonard Kleinrock. *Queueing Systems*, volume II: Computer Applications. John Wiley & Sons, 1976.

[7] Anita Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 15450 S.W. Koll Parkway, Beverton, OR 97006, 1986.

[8] Mark K. Seager. On the synchronization requirements of parallel supercomputer applications. *in preperation*, 1989.

[9] Mark K. Seager, Robert E. Strout, Nancy E. Werner, and Mary E. Zosel. Graphical analysis of multi/micro-tasking execution on cray multiprocessors. Technical Report UCRL-100267, Lawrence Livermore National Laboratory, PO Box 808, Livermore, CA 94550, 1988.

[10] Richard Watson. The architecure of future operating systems. In Karen Winget, editor, *Cray User Group Incorporated 1988 Fall Proceedings*, pages 203–208, 1988.

A Sample Simulator Input and Output

In most input files only the parameters that need to be different from the defaults are changed. The following is a sample input file that for an eight processor system that has one half of the jobs being serial and one half being parallel with probability $\frac{1}{8}$ that they will have 2, 4, 6 or 8 processes in them. All units are in micro-seconds.

```
#       General simulation parameters...
SchMethod       = 2             # for GANG scheduling
SimLength       = 1.0E9
OutputDelta     = 1.0E8
GenMethod       = 0             # Keep MinProc jobs in the system...
MinProcsInSystem = 24           # Number of active processes.
#       Job Generation criterion...
#       Should look like SIMPLE...
SIMMean         = 4090.0        # Mean work between barriers
SIMStdDev       = 409.0         # StdDev is 10% of SIMMean
SISMean         = 0.0           # Noise has zero mean
SISStdDev       = 204.5         # 5% of SIMMean
NBMean          = 122           # Mean number of barriers per job.
NBStdDev        = 90
#                Serial 2     3     4     5     6     7     8
ParArray        = 0.500 0.125 0.000 0.125 0.000 0.125 0.000 0.125
```

The above input file generated a great deal of output and the following is only the initial parameters and the final `Short-term statistics` and `Long-term statistics` output. The run was made on a Sun 3/160+FPA.

```
DEBUG               = 0
SchMethod           = 2
GenMethod           = 0
NumCPUs             = 8
MinProcsInSystem    = 24
MinLoad             = 3.000000
DelayMean           = 500000.000000
SIMMean             = 4090.000000
SIMStdDev           = 409.000000
SISMean             = 0.000000
SISStdDev           = 204.500000
NBMean              = 122.000000
NBStdDev            = 90.000000
GlobalTimeSlice     = 100000.000000
GlobalSpinWaitDelay = 1000.000000
GlobalOverhead      = 350.000000
SimLength           = 1000000000.000000
OutputDelta         = 100000000.000000
RandomSeed          = 0
ParArray            = 0.5000 0.1250 0.0000 0.1250 0.0000 0.1250 0.0000 0.1250
Repeatable random seed: 620082279

Fri Aug 25 15:04:45 1989
Current Time 1.000000e+09 Cpu Time 3.33e+09 Ratio 3.00e-01 Load 3.23e+00

Short-term statistics from time 900000954.300896 to time 1000000009.050145:

Job and Process Completion (Throughput) Statistics
        ProcsInJob       JobsComp         ProcsComp
STTP             1            213               213
STTP             2             60               120
STTP             4             45               180
STTP             6             60               360
STTP             8             58               464
STTP         Total            436              1337


                Average Per-Process Statistics
          MiliSec MiliSec MiliS MiliS MiliSec        Turn    Cntx
```

```
      Proc ReadyQu UserWrk   Sys SpinW BlockQu Ovrlp  Around Swtch
STAPP   1     656     588  4.31     0       0  1.00    1248  12.3
STAPP   2    1534     561  9.15    12      58  1.85    2175  26.1
STAPP   4    2177     498 10.68    19     242  3.43    2949  30.5
STAPP   6    1338     557  4.93    25     119  5.58    2046  14.1
STAPP   8    1089     574  4.69    37      60  7.82    1765  13.4
STAPP Tot    1274     560  5.90    23      91  5.00    1954  16.9

               Standard Deviation Per-Process Statistics
           MiliSec MiliSec MiliS MiliS MiliSec         Turn   Cntx
      Proc ReadyQu UserWrk   Sys SpinW BlockQu Ovrlp  Around Swtch
STSDP   1     560     344  4.21     0       0  0.00     821  12.0
STSDP   2    1099     322  8.40    11     103  0.14    1409  24.0
STSDP   4    1857     273 14.00    14     503  0.66    2292  40.0
STSDP   6     853     340  4.30    22     250  0.45    1229  12.3
STSDP   8     643     335  6.18    35     140  0.20    1044  17.7
STSDP Tot    1073     330  7.68    27     253  2.60    1418  21.9


               Queue Length Statistics
                          Queue           Avg            StdDev
STQ   Low-priority Ready Queue          11.66              4.30
STQ  High-priority Ready Queue           5.40              3.52
STQ               Blocked List           1.23              1.79
STQ               Load Average           3.13              0.35

               CPU Utilization
STCPU   Idle Time:                    5005047.42   0.63%
STCPU   User Work Time:             755777109.40  94.47%
STCPU   System Overhead:              7966350.00   1.00%
STCPU   Spin Wait Time:              31243931.17   3.91%


Long-term statistics from time 0.000000 to time 1000000009.050145:

Job and Process Completion (Throughput) Statistics
        ProcsInJob       JobsComp         ProcsComp
LTTP             1           2250              2250
LTTP             2            573              1146
LTTP             4            532              2128
LTTP             6            550              3300
LTTP             8            565              4520
LTTP         Total           4470             13344


               Average Per-Process Statistics
           MiliSec MiliSec MiliS MiliS MiliSec         Turn   Cntx
      Proc ReadyQu UserWrk   Sys SpinW BlockQu Ovrlp  Around Swtch
LTAPP   1     651     567  4.26     0       0  1.00    1222  12.2
LTAPP   2    1385     560  8.40    11      58  1.85    2023  24.0
LTAPP   4    2223     570 11.17    21     244  3.37    3071  31.9
LTAPP   6    1325     570  4.63    27     110  5.66    2037  13.2
LTAPP   8    1070     562  4.51    36      55  7.76    1727  12.9
LTAPP Tot    1273     566  5.89    23      90  4.89    1958  16.8

               Standard Deviation Per-Process Statistics
           MiliSec MiliSec MiliS MiliS MiliSec         Turn   Cntx
      Proc ReadyQu UserWrk   Sys SpinW BlockQu Ovrlp  Around Swtch
LTSDP   1     578     325  3.85     0       0  0.00     817  11.0
LTSDP  2     984     321  6.82   11     119  0.15    1284  19.5
LTSDP  4    1760     339 13.06   20     453  0.67    2268  37.3
LTSDP  6     850     326  4.65   27     261  0.49    1226  13.3
LTSDP  8     599     331  4.62   32     111  0.42     956  13.2
LTSDP Tot   1076     329  7.26   28     246  2.62    1437  20.7


               Queue Queue Length Statistics
                        Queue         Avg          StdDev
LTQ   Low-priority Ready Queue       11.53           4.37
LTQ  High-priority Ready Queue        5.44           3.56
LTQ               Blocked List        1.20           1.73
LTQ               Load Average        3.12           0.34

    CPU Utilization:
LTCPU   Idle Time:                 48313127.69   0.60%
LTCPU   User Work Time:          7563416447.80  94.54%
LTCPU   System Overhead:           78831550.00   0.99%
LTCPU   Spin Wait Time:           309438946.91   3.87%


                  Histogram: Processes per job
  8.500:              |                                                   0.00%
  7.500:      8.500 |*************                                      12.67%
  6.500:      7.500 |                                                    0.00%
  5.500:      6.500 |************                                       12.31%
  4.500:      5.500 |                                                    0.00%
  3.500:      4.500 |************                                       11.91%
  2.500:      3.500 |                                                    0.00%
  1.500:      2.500 |*************                                      12.80%
  0.500:      1.500 |************************************************* 50.30%
       :      0.500 |                                                    0.00%
                    +-------------------------------------------------
                    00000000000000000000001111111111111111111111222222
                    00011223344555667788990011122334455667778899001122
                    04938372616059493827261505948382716150494837271605
                    06284062840628406284062840628406284062840628406284

                  Histogram: Percentage of spin wait used
100.000:              |                                                   0.00%
 90.000:    100.000 |*******                                             4.83%
 80.000:     90.000 |***                                                 1.83%
 70.000:     80.000 |****                                                2.48%
 60.000:     70.000 |*****                                               3.37%
 50.000:     60.000 |*******                                             4.70%
 40.000:     50.000 |*********                                           6.37%
 30.000:     40.000 |*************                                       9.14%
 20.000:     30.000 |******************                                 12.81%
 10.000:     20.000 |***************************                        19.34%
  0.000:     10.000 |************************************************* 35.13%
       :      0.000 |                                                    0.00%
                    +-------------------------------------------------
                    00000000000011111111111122222222222233333333333344
                    00123455678900123455678900123456678901123456678901
                    08653108753209754219764319865310865320875420976421
                    03715937159371582604826048260371593715937158260482
                    09877655433210098876654332110998766544322109987765
                    03692581470369258147036925814703692581470369258147
                    Histogram: Percentage of time slice used
100.000:            |********                                          5.93%
 90.000:   100.000  |*******************************************      31.36%
 80.000:    90.000  |***                                               1.81%
 70.000:    80.000  |***                                               1.71%
 60.000:    70.000  |***                                               1.84%
 50.000:    60.000  |***                                               2.00%
 40.000:    50.000  |****                                              2.36%
 30.000:    40.000  |*****                                             3.04%
 20.000:    30.000  |*******                                           4.70%
 10.000:    20.000  |*************                                     8.95%
  0.000:    10.000  |************************************************ 36.31%
       :     0.000  |                                                  0.00%
                    +------------------------------------------------
                    0000001111112222223333334444445555556666667777788
                    0135680135680135680135680135680135680135680135680
                    0630630630730730730740740740741741741741841841841
                    0741852963074185296307418529630741852963074185296
                    0000000000000000000000000000000000000000000000000

                    Histogram: Size of system owners
  8.500:                    |                                          0.00%
  7.500:     8.500  |************************************************ 34.29%
  6.500:     7.500  |                                                  0.00%
  5.500:     6.500  |**********************************************   32.93%
  4.500:     5.500  |                                                  0.00%
  3.500:     4.500  |***********************************              24.88%
  2.500:     3.500  |                                                  0.00%
  1.500:     2.500  |**********                                        6.68%
  0.500:     1.500  |**                                                1.22%
       :     0.500  |                                                  0.00%
                    +------------------------------------------------
                    0000000000000011111111111112222222222222333333333
                    0012334566789901223455678890112344567789001233456
                    0752075207520752075207520752075207520752075207520
                    0505050505050505050505050505050505050505050505050

                    Histogram: Size of cycle suckers
  8.500:                    |                                          0.00%
  7.500:     8.500  |*                                                 0.68%
  6.500:     7.500  |                                                  0.00%
  5.500:     6.500  |****                                              3.57%
  4.500:     5.500  |                                                  0.00%
  3.500:     4.500  |***********************                          23.30%
  2.500:     3.500  |                                                  0.00%
  1.500:     2.500  |*********************                            21.16%
  0.500:     1.500  |************************************************ 51.29%
       :     0.500  |                                                  0.00%
                    +------------------------------------------------
                    0000000000000000011111111111111111222222222222222
                    0011223345566778890011223345566778890011223345566
                    0516273840516273940516283940517283940617283950617
                    0517284951728495162849516283951628395062839506273
                    0741852963074185296307418529630741852963074185296
```