Portable Batch System

Mark A. Smith, Arvind Santhanam, Jamison Collins.

Final Project

CSE 221

I. Introduction

Computing performance is crucial in modern scientific environments. However, a distinction must be drawn between a high performance computing environment, and a high throughput computing environment. The former is primarily concerned with the latency to complete the execution of a single program while the latter is concerned with throughput, or the total time to execute *several* programs. While a single extremely fast machine can achieve both a high performance and a high throughput environment, the price of such an environment will be prohibitive. Thus, high throughput can be better achieved by making use of several cheaper machines, and executing several jobs on them in parallel. However, it is important to properly manage these jobs in order to maximize throughput. A simple scheme may be easier to implement, but it also may result in a heavy load imbalance, in which most machines are idle while a few execute most of the jobs. This work is concerned with the general approaches to managing jobs on several machines, and the specific approach taken in the High Performance Compilation and Architecture Laboratory at UCSD.

II. Current Lab Environment

There are two primary ways jobs are run in the lab: either a fully static division in which the user writes a shell script for each machine on which jobs will be run, or using the PQS(Perl Queue System). The primary disadvantage of the first scheme is that the user is forced to decide on the partitioning of jobs to machines. If the user fails to partition the jobs evenly or external circumstances(such as high priority interactive jobs) interfere, the result will be an imbalance in which, after most jobs have been completed, many machines are idle while a few still have several more jobs to run. Thus, the script method may fail to achieve good load balance even when only a single user has jobs to run.

Under PQS, the user runs an instance of the PQS daemon on every machine, and places a set of job descriptors into a special directory. Each of these daemons independently probes the directory every minute to check for the presence of a job to run; if one is detected and the daemon is not already executing a job, the new job is locked and execution begins. Through this competetive approach, PQS assigns the next job to the first available processor, ensuring that a user has one job running on all machines whenever a sufficient supply of jobs is available. However, because there is no central scheduling daemon, only local scheduling policies such as this can be enforced. In fact, because one instance of the daemon has no knowledge of any other instance (belonging to this user or another), PQS is unable to make any load guarantees–it is possible (and common) for jobs from several users to be executing on one machine while another machine is idle. Additionally, by greedily assigning jobs, a slow machine may be given a job to run even though a faster machine is available.

Additionally, neither of the existing schemes perform well when one user is given a higher priority over others, which occurs frequently before a paper deadline. Typically, several machines will be set aside for those priority users, or the high priority user will run his jobs at an elevated priority level, which may cause the lower priority user's job to cease making progress for a long time. Clearly both of these approaches are unsatisfactory solutions.

PQS outperforms the script based system becauase it can achieve a reasonable (though definitely suboptimal) assignment of jobs to machines when only one user has jobs to run, however it can perform poorly when several users have jobs to run.

In order to address these problems, a more advanced batch system must be used. Large scale batch systems can overcome the above problems through the use of a central scheduling daemon and an advanced scheduling policy specification language(such as C). Global knowledge can insure that jobs are migrated whenever there is a load imbalance, or a higher performance machine is idle and a slower one is running a job. Finally, user priorities can be cleanly enforced. This can be done passively, by not scheduling jobs from a lower priority user if he already has too many jobs running and high priority jobs are queued, or aggressively, by halting or migrating lower priority jobs when a higher priority user has jobs to run.

Of the freedoms provided by an advanced batch system, for this lab the most important are the ability to achieve a fair use of the machines, to effectively implement a user priority scheme and to effectively deal with the heterogeneous structure of the lab. Under PQS, users who need to run fewer jobs are penalized. For example, if user A only needs to run eight jobs, and user B needs to run 100, then user A will have to share his eight machines with user B. This is a poor scheduling configuration because it will greatly increase the turnaround time for the user with eight jobs, while having only a small improvement in turnaround time for the user with 100 jobs. The user priority issue, as laid out above, must be effectively dealt with in order for a batch system to be useful in this lab. Many existing batch systems are able to manage some form of user level priority scheme, but the actual scheme implemented (described later) could not be implemented by the default scheduler on existing systems. Finally, because the fastest machines in this lab are more than twice as fast as the slowest machines, it is important to consider this in order to improve overall turnaround time.

The PBS was deployed at UCSD as part of this work. There were two significant deviations from the standard installation, the scheduler used and the incorporation of process migration. The scheduling policy implemented was a nonstandard policy in which jobs were scheduled such that users recieve an amount of computational power propotional to their priority, divided by the number of users with currently queued jobs. For example, if two users with priority 1 and one user with priority 3 have jobs queued, the scheduler would attempt to give the first two users 20% of the power and the third user 60%. This is achieved in a somewhat passive manner, with the scheduler making a decision as to which job to schedule in order to bring the overall load closer to this goal. Each machine only runs one job at a time. These percentages will not translate directly into a percentage of processors because of the heterogenous composition of machines in the lab. As stated, process migration was also incorporated into PBS. Because PBS lacked the necessary capabilities, it was necessary to make use of the libraries provided with Condor to enable process migration. As it currently stands, the scheduler will migrate a process from a slower machine to a faster one when it detects that a faster processor is currently idle. There are certainly more advanced uses of this capability that could be implemented, and work on the scheduler will continue throughout the summer before widespread use begins next Fall.

III. PBS Vs Condor

Given the present state of the software in the architecture lab and the specialized scheduling needs its type of processes demand, we were faced with the challenge of finding an appropriate software system. After performing a fairly intensive search, we found two main systems which had features in which we were interested. First, a system called Portable Batch System (PBS) [1][2][3], a project with avaiable source code and customizable scheduling, and second a project from the University of Wisconsin, Madison named Condor [4-7], a more rigidsystem, but that offered more advanced features such as built-in process migration.

PBS, as its name implies, is a batch system that utilizes three main processes to schedule and run a set of processes. First is the server, which detects events such as job termination or job submission and informs the scheduler. Second is the scheduler which, in response to events, chooses the next job to run. Finally, one instance of the ”Mom,” process runs on each machine. It is responsible for actually executing the job it is assigned to run by the scheduler, and to send any results back to the server. The most attractive feature of PBS, however is the ability to customize the scheduler. This is important because the default scheduler, while powerful and customizable, was targetted toward an environment which differed greatly from the Architecture Lab. It was important to insure that the behavior of the scheduler could be fine-tuned in order to provide exactly the policy demanded by this environment. The flexibility of the scheduler is a great attraction, but unfortunately PBS does not provide process migration.

Condor, on the other hand, offers built-in process migration, a desirable feature for the jobs the architecture lab runs. Condor has a slightly different structure than PBS. Each machine in a Condor pool runs three processes. First, ”schedd,” the scheduling deamon, which keeps track of all the jobs submitted to a particular machine. Second, ”startd,” which is in charge of monitoring the machine on which it is running to determine whether it is available to run a Condor process. It uses information such as mouse and keyboard activity, and CPU load to make this decision. Finally, the ”master” process makes sure that schedd and startd continue running and if they terminate too often, it notifies the administrater through email. There is one machine in a pool which runs two extra processes, the ”collector,” and the ”negotiator.” The collector is responsible for receiving all information from each of the schedd and startd deamons running in the pool. The negotiator uses information available from the collector to schedule jobs on machines. The negotiator policy, much like PBS’s default scheduling policy, is powerful and adjustable, however differs from PBS’s scheduler in one critical way: it is not fully customizable. A set of scheduling policies can be constructed from building blocks, however, as before, critical fine-tunining is critical to provide exactly the policy demanded by the Architecture Lab’s environment. Regardless of the policy, the mechanism is the same. When a job is submitted to a machine, the schedd daemon starts a ”shadow” which acts as the liason between the running process and the submitting machine. In order to actually start the job, the startd daemon creates a ”starter” process that is in charge of submitting, monitoring, checkpointing, and stopping a process. The feature of process migration is very desirable, but unfortunately, the negotiator scheduling policy is not modifiable to the degree we had hoped. Further, members of the lab had experience with PBS's interface, while Condor would require training.

One last feature that is important to have is ease of scalability. Adding and removing machines from the lab should not cause a major rewrite or reconfiguration to occur. With present scripting or PQS as described, adding or removing a machine requires rewriting scripts and reconfiguration. It is important that in the dynamic computing environment in the lab, that adding and removing machines be simple. In both PBS and Condor, near optimal scalability is achieved without much reconfiguration. In PBS, the administrator must notify the server that a new machine is on the network and run the ”Mom” process on that machine. The scheduler then adds that machine to its possible availability list and uses that machine's ”Mom” process as it would on any other machine. Condor is very similar. The new machine must run the ”master” process and the ”collector” be notified.

Given these two systems that were equal in most respects, we were faced with a dilemma. Should we opt for the familiar system with the fully customizable scheduler, a critical need, but that had no process migration capabilities; or should we choose Condor, which offered process migration, but would not suit the lab's scheduling needs. Given the conversation at our project meeting, there was one final variable we had to weigh. Did we want to pass OS? It seemed clear that getting process migration working was an important key to unlocking the door that led out of OS successfully. In traditional computer science style, we opted for a hybrid. We discovered that it was possible to migrate processes by simply linking with Condor's libraries and then sending the running process a signal at which point it would write a checkpoint file and terminate. The checkpoint file could be read in from another machine and the process restarted on the new machine. PBS offered the capability of sending running processes signals. Combining these features of both systems, we were able to achieve all of our goals: getting a system running in the lab which would be used, and passing OS. Given the described hybrid software system, we wanted to produce some measure of its effectiveness. Results of sample runs are presented in the next section.

IV. RESULTS AND DISCUSSION

A scheduling policy is a mechanism which resolves contention of resources amongst the contending parties. One of the main goals of a scheduling policy is to minimize the total turn-around time of jobs in the system while maximizing system utilization (SU). The scheduling algorithms for a batch system can try to optimize on achieving high system utilization in a local and global way. A local optimization technique (greedy approach) would essentially place the current job on the fastest available server. For global optimization, the scheduler would evaluate the load on each of the workstations in the cluster and place the job on the machine which would result in highest system utilization. This however would require detailed state information of each of the workstations and the nature of the jobs ready to be scheduled.

We ran a set of experiments to gain insight into what scheduling policy might be best suited for our work environment. The various scheduling policies we implement are:

1. SCRIPT: Each user allocates which machine he wants to run his jobs on. The system just executes the jobs on those machines as desired by the user.
2. FAST: The current job is scheduled on the fastest available server in the cluster, and remains there until completion.
3. MIGRATE: A job on a slow machine is migrated to the faster machine.
4. P-MIGRATE: Migrates the highest priority process to the fastest available server. There are 3 priority levels, *high, medium* and *low*.

We simulate the aforementioned scheduling policies for a single user and a multi-user environment. In case of a single user, all the resources are available at the user's disposal. However, in the event that multiple users run jobs in tandem, there arises contention for resources. The scheduling policy must resolve these contentions while being fair to each user. We resolve this by allocating priorities to each user and scheduling the job with the highest priority. This would ensure that the user has a fair share of system resources commensurate with his priority. In our simulation environment the priorities for each of the jobs remains the same throughout the run, unless explicitly changed by the user.

We were interested in simulating an algorithm in which each workstation maintains a queue of jobs, a job being scheduled on the machine with maximum amount of available resources normalized by the processing speed, but implementing process migration in such a context although interesting, seemed tangential to the needs of the lab in which all processes are CPU intensive.

A. Simulation Environment

The simulation configuration comprised of *fourteen 21164s, four 500 MHz 21264* and *four 667 MHz 21264* workstations connected as a token-ring configuration. We ran the 18 spec benchmarks on one of each kind of machine. The execution times of these benchmarks gave us valuable insight as to the relative speeds of these machines with respect to each other. The 500MHz 21264 was twice as fast as the 21164, but was only 0.75 times as fast as the 667MHz 21264 machine.

We compare various policies by running a large set of jobs on the system. One measure that reflects system performance is the amount of resources wasted (WR) by the system. The wasted resources (WR) is computed by measuring the weighted SUM of DIFFERENCES between the finish time of the last machine and the other machines in the system. Another useful measure is the *difference in the finish times of the first released machine and the last*. We denote this by XR. Finally, the turn-around time (TAT) of jobs in the system can be used as a measure to directly reflect the performance of the algorithms.

A large number of jobs were run on this testbed for SCRIPT, FAST, MIGRATE and P-MIGRATE. The degree of *burstiness* of jobs was set to be fairly high, so that the simulation results would reflect the performance of the scheduling policies in plausible worst-case scenarios. High burstiness of jobs also gives us some valuable insight as to how gracefully service degrades for each of these policies. The size of the jobs were uniformly distributed over a small range. The advantages of doing this were two-fold. First, it helped alleviate the skew in the measured parameters XR and WR, since these would have a strong dependence on the size distribution of the jobs. Secondly, it was able to reflect the randomness in the sizes of jobs that might be present in the system at any given time.

B. Observations

1. **Single User Environment:** SCRIPT performs poorly due to static allocation of system resources and lack of system-level optimization. SCRIPT is an absolute user-defined program and so it would be unfair to compare it with other scheduling policies. FAST performs drastically better as the scheduler employs a greedy approach of usurping the

fastest available server. A notable feature of FAST is that it tries to keep every machine engaged as much as possible, consequently has very low XR and WR. MIGRATE outperforms FAST as it is able to transport processes running on slower machines onto faster machines, thereby having a low WR quotient. MIGRATE has a slightly higher XR, since the faster machines are busy until the end while the slower machines relinquish their processes early on to faster machines. However, MIGRATE has a lower WR quotient than FAST, suggesting the importance of process migration. P-MIGRATE is discussed in the Multi-user Environment section.

Also, the key measure of performance, TAT, for MIGRATE is faster than FAST. FAST, however is the optimal strategy in a scenario with no process migration. Another measure of interest is the average time (AT) it would take if all the jobs are scheduled in the first-come-first-served order (FCFS) on a single machine which has the processing power of the entire system and infinite memory bank. We find that the execution times for MIGRATE converges faster to the estimated AT, than FAST.

A salient feature in a large number batch systems is that migrating processes adds tolerable overhead in comparison to the size of the jobs being simulated. It was observed that many of the jobs run in the lab had high turn-around times of 30-45 minutes on an average, while the average time to migrate a process was close to five seconds. This is a best case scenario where one can tolerate a high degree of process migration. However, if we have a large number of small processes migrating in the system, the performance might degrade due to high communication overhead. A plausible solution in such a scenario would be to have a copy of all the active processes in the system on all the machines (stored in disks). On a Checkpoint, only a minimum amount of state information need be transfered to keep the communication overheads low. This scheme is suitable in a set-up with large disks, small-sized tasks, and where using extensive process migration might pay great dividends.

Algorithm	$\frac{XR}{MAX-JOB}$	$\frac{WR}{WR_{FAST}}$	$\frac{TAT}{TAT_{FAST}}$	$\frac{(TAT-AT)\times 100}{AT}$
FAST	0.726	1.0	1.0	11.5
MIGRATE	0.755	0.733	0.92	2.3
P-MIGRATE	0.742	0.798	0.94	4.7

$MAX - JOB$ is the maximum size of a job. WR_{FAST} and TAT_{FAST} are the wasted resources and turn-around times for the FAST algorithm respectively.

2. **Multi-User Environment:** In the eventuality when all jobs running in the system have the same priority the multi-user system emulates the behaviour of a single user system. The system schedules processes in order of priorities. Each process has a priority which is assigned by its owner. Priority assignment ensures fair share of resources. However, as pointed out earliar, priority assignment must be monitored by the system so that no single user can maliciously trigger a large set of processes of high priority, thereby affecting the conformant users. So the system must assign priorities giving due consideration to the user desired priority as well as the number of tasks he decides to run.

P-MIGRATE permits fair share of resources assuming there is no malicious user in the system. In the simulation run, we priority distribution was as follows: 10% high priority jobs, 40% medium priority jobs and 50% low priority jobs. P-MIGRATE had a higher TAT and WR quotient than FAST but lower than MIGRATE. This amply demonstrates that process migration helps in decreasing TAT. However, use of priorities decreases the flexibility of the scheduler to schedule processes, unlike MIGRATE which transports process without any discrimination from a slower machine to a faster machine, on availability. This is the reason MIGRATE outperforms P-MIGRATE. However, it was observed that the high priority jobs had a lower average TAT than the average TAT for a job scheduled by MIGRATE, in accordance to our expectations.

In modern batch systems, it is very common to implement time-slicing strategies wherein a job is executed for a certain duration and context-switched with another process with equal or higher priority. Using a round-robin mechanism to schedule jobs would ensure fairness at the granularity of the time-slice.

V. Conclusion

Batch processing is a crucial element in the realm of high performance computing. In this paper, we shed light on some important aspects of batch processing systems in existence today, and in particular PBS enhanced with Condor's process migration facility. A variety of scheduling policies were implemented on PBS, and a performance evaluation for each of these schemes were drawn. These results suggested the importance of process migration as an integral part of user or system defined scheduling policies. However, the trade-off between degree of process migration and simplicity of implementation must be weighed carefully. Engineering an approach for batch systems is often dependent on the available hardware support and the requirements by the end-users. This emphasizes the need for scalability and easy re-configurability of the batch processing system.

REFERENCES

[1] A. Bayucan, R.L. Henderson"Portable Batch System," *Technical Report http://pbs.mrj.com,* November 1999.

[2] J.P. Jones, "NAS Requirements Checklist for Job Queuing/Scheduling Software," *NASA Technical Report*, NAS-96-003, April 1996.

[3] R. L. Henderson, ","Job scheduling under the Portable Batch System. *Job Scheduling Strategies for Parallel Processing. IPPS'95 Workshop. Proceedings,* April 1995.

[4] Livny, M. and Mutka, M. W., "The Available Capacity of a Privately Owned Workstation Environment," *Performance Evaluation,* vol. 12, no. 4 pp. 269-284, July, 1991.

[5] Litzkow, M., Livny, M., and Mutka, M. W., "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems,* pp. 104-111,June, 1988.

[6] Litzkow, M., "Remote Unix - Turning Idle Workstations into Cycle Servers," *Proceedings of Usenix Summer Conference,* 1987.

[7] http://www.cs.wisc.edu/condor/project.html