Job Scheduling for Parallel Processing

Asst. Prof. Shubhada Talegaon
Parul Institute of Engineering And Technology

Abstract-The important topic in parallel computing is job scheduling. Parallel computing systems such as Supercomputers are valuable resources which are commonly shared among each member of a community of users. The main concern of scheduling is how to share the resources of parallel machines among the number of competing jobs, giving each the required level of service and maximize system's utility. Effective scheduling strategies to improve response times, throughput, and utilization are an important consideration in large supercomputing environment. In this paper, I present the analysis of various job scheduling techniques such as "Space sharing", "Back filling" and "Gang scheduling". I have even discussed how grid computing and certain current trends in parallel processing improve the performance.

Index terms- Back filling, Gang Scheduling, Grid computing, Job Scheduling, Space Sharing

1. INTRODUCTION

Parallel supercomputers are expensive, scare recourses that often must be shared with community of users. The resource allocation for the competing jobs is done by system scheduler. The scheduling on parallel computer is complex since it involves scheduling over two dimensions- time and space and two levels- jobs and thread. The large verity of parallel programming languages, parallel architecture and parallel operating systems mean that there is no universally accepted job scheduling strategy for parallel system.

Jobs are continually submitted by users to the system, each with unique resource and service-level requirements, some with large batch jobs and others with small interactive jobs. Proper scheduling and resource allocation becomes critical issue. Scheduling is an inherently reactive discipline, mirroring trends in High Performance Computing (HPC) architectures, parallel programming language models, user demographics and administrator priorities. No scheduling strategy is optimal for all of today's scenarios.

In recent years, relative stability in the aforementioned forces has gradually moved large supercomputer installations towards workable though imperfect de facto standards. The production of large Massively Parallel Processing (MPP) machines today centers around Multiple Instruction Multiple data (MIMD) architectures in pure or shared distributed memory configurations, such as Cache Coherent Non-Uniform Memory Access. This architectural trend has given rise to the supremacy of rigid programming models such as Message Passing Interface (MPI) and consequently of complementary scheduling policies such as Batch queued space-sharing and its variants. Concurrently, the rigidity and explicit parallelism of MPI is slowly giving way to alternative programming models which challenge traditional scheduling.

In this paper, I illuminate the issues and approaches that have defined how parallel jobs are scheduled in today's production environments.

2. TERMINOLOGY:

- **Job:** Jobs are autonomous program that execute in their own protection domain. It is composed of multiple concurrent threads submitted to the system for execution. Each job is characterized along two dimensions: its **length** as measured by execution time and its **width** or *size* as measured by the number of threads; assumption is that each of a job's threads executed on a separate processor.

IJSER © 2013
http://www.ijser.org

- **Job scheduling:** A discipline whose purpose is to decide when and where each job should be executed from the perspective of the computing system.

- **Time Sharing:** It refers to any scheduling approach whereby threads can be preempted by others during execution and restarted later.

- **Multi programming level:** The number of jobs that each processor can execute concurrently is known as the m*ultiprogramming level*.

- **Space-Sharing:** Space-sharing approaches provide a thread exclusive use of a processor until its execution is complete or a maximum time limit has been over and the thread is terminated. It manages time by placing each job in a queue and executing all of its threads concurrently upon release from that queue.

- **Interactive jobs:** It require low latency are usually executed using time-sharing

- **Batch jobs:** That require unperturbed performance and are executed on dedicated processors using space-sharing

Supercomputing facilities often meet the requirements of both Interactive and Batch job categories by statically partitioning a machine's processors into time-sharing and space-sharing subsets.

3. METRICS USED FOR EVALUATION

Parallel job schedulers are mostly evaluated using performance metrics, fairness matrix, and Predictability metric.

Performance Metric:

A performance metric is a representation of how quality of service from the system is interpreted. The metric can be *system based* or *user based*. System based metrics include utilization and capacity loss. Common user based performance metrics include average waiting time (AWT), average response time (ART), average job slow-down (AJSD) and throughput.

Fairness Metric:

Most scheduling algorithms make the minimum fairness guarantee that no job will be starved, that is, each job will eventually execute. Stronger fairness guarantees are contingent on the scheduling scheme. In space-sharing, fairness may imply some first-come-first-serve (FCFS) ordering or that a job will not be delayed by any job that is behind it in the queue. In time-sharing, it may be that each thread receives an equal slice of the processor or a slice weighted by the job size.

Predictability Metric:

Predictability is the gap between a job's responses or flow time and the user's expectation as created through previous experience. Predictability can indirectly increase productivity by enabling users to anticipate job completion times and plan resource usage accordingly. Some have proposed that predictability, under other realistic assumptions, may be even more central to the user experience than performance.

IJSER

4. JOB SCHEDULING ON MPP SUPERCOMPUTERS

The dominant resource for parallel processing in recent years has been the MPP supercomputer. In this section I have focused on some ideas and issues related to job scheduling on MPP supercomputers.

4.1 Space-Sharing

The simplest way to schedule a parallel system is with a queue. Each job is submitted to the queue and, upon reaching the head, is executed to completion while all other jobs wait. The queue can be hypothetically FIFO, but the scheme extends to priority queues without loss of generality. Though providing maximum fairness and predictability, this scheme is inefficient. Since each application utilizes only a subset of the system's processors, those processors not in the subset are left idle during execution. This effect is known as **fragmentation** and its reduction is the primary focus of much scheduling research. The most natural extension to the queue scheme is space sharing, which is the simple idea of allowing another job in the queue to execute on the idle processors if enough are available. This is primarily how supercomputers are scheduled today.

4.1.1 Backfilling

The most basic queued space-sharing approach is known as blocking First-Come-First-Serve (FCFS). Under this scheme, if sufficient idle processors exist to serve the next job in the queue, that job is executed. Otherwise, the queue blocks until sufficient resources become available. This approach remains prone to severe fragmentation with system utilization rates between 50-80%. Because the queue is only accessed at the head, a wide job may block others behind it from executing while it waits for a large portion of the machine to become available.

Backfilling is the idea that while the wide job waits, the scheduler may choose to execute some narrower jobs situated further back in the queue. The question is, which job should jump ahead? The first implementation of a backfilling scheduler was the Extensible Argonne Scheduling sYstem (EASY). The scheduler was deployed on the Argonne National Laboratory's 128-node IBM SP system and was successful enough to be eventually incorporated into IBM's commercial LoadLeveler scheduling software. EASY backfilling, as it has come to be known, works by allowing a narrower job J*n*, to jump in front of a waiting wide job J*w*, so long as the execution of J*n* does not delay the projected start of J*w*. The job furthest ahead in the queue that satisfies these width and length requirements is selected for backfilling. This scheme relies on a significant assumption, that is, that job lengths are known a priori. Argonne's approach, which preponderates today, was to simply ask the users for an expected runtime. Though this approach has proven serviceable, the problem of job length estimation under disparate assumptions has created an active field of research as described in Section 4.1.2. The other problem with EASY backfilling is fairness, as cutting can cause unfairness even if not to the job at the head of the queue. This is the fundamental observation motivating *conservative backfilling*.

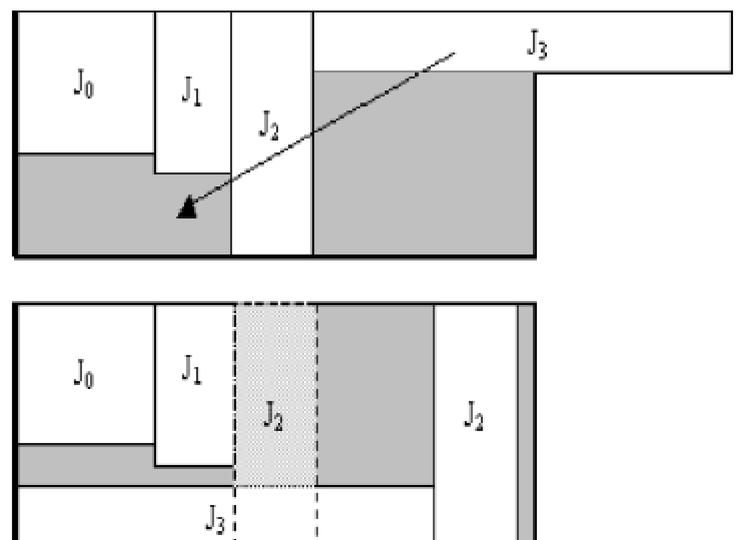

Figure 1 EASY backfilling cause unfairness

Figure 1 demonstrates how this unfairness can occur. In the figure, jobs are ordered from left to

right and the Y axis represents the number of processors. Job J0 is currently executing, leaving too few processors to execute J1. The first job that can satisfy the length and width requirements of the EASY backfilling algorithm is J3, so it is scheduled. However, though it has no effect on J1, the execution of J3 delays the start time of J2. Conservative backfilling only backfills when the scheme causes no job to be delayed.

The tradeoffs between conservative and EASY backfilling can be generalized to a *number of reservations*, where EASY backfilling makes a single reservation for the job at the head of the queue and conservative backfilling makes one for each job in the queue.

4.1.2 Estimating Job Lengths

As mentioned in the previous section, backfilling is predicated on knowledge of the job lengths before execution. The approach taken by Lifka's EASY scheduler, to simply ask the users to submit an expected runtime along with the job, is in wide use today. Unfortunately, estimates gathered in this manner are notoriously inaccurate. Indeed, inaccurate runtime estimates have profound effects on fairness and predictability. Wildly exaggerated runtime estimates result in greater system throughput. This happens because the premature termination of jobs causes fragmentation in the schedule. In backfilling, that fragmentation is ease by executing shorter jobs from the back of the queue. It is no surprise that this arrangement increases performance as scheduling theory has long recognized that the Shortest Job First (SJF) heuristic results in optimal throughput SJF is not widely used on today's production installations because of its inadequate fairness.

Erratic runtime estimates can also manifest unfairness through a phenomenon known as *pseudo-delay*. An example is shown in Figure 2. Job *J*1 is prevented from executing at the same time as *J*0. Relying on false runtime estimates, the scheduler decides to backfill *J*2. Soon thereafter, *J*0 completes execution, but *J*1 cannot begin because of the decision to backfill *J*2. The backfilling fairness guarantee that *J*1 would not be delayed by any job behind it in the queue is broken.

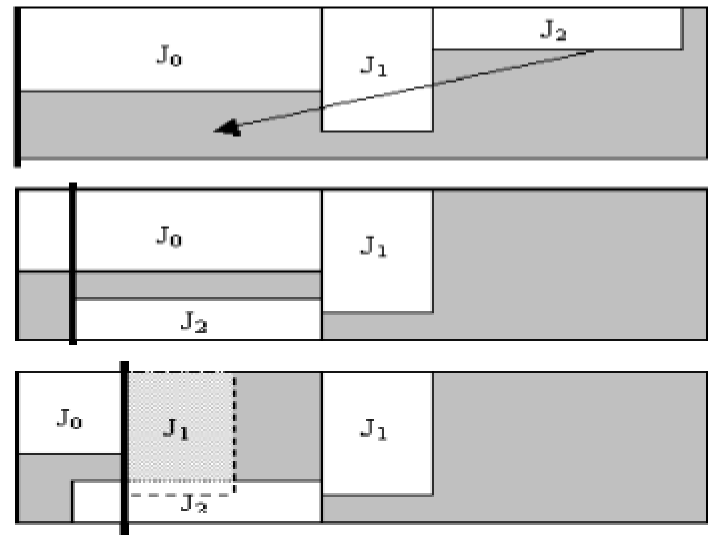

Figure 2: Example of pseudo-delay

Like fairness, system predictability also suffers because of poor runtime estimates.

4.1.3 Predicting Queue Times

Queue time predictability can be correlated with productivity through many scenarios. The most obvious example is a user with accounts on numerous machines who wants his job to finish most quickly. More subtle is the lost utility caused when the user's job does not finishing in time to provide useful output (e.g. a one day weather simulation finishing in twenty five hours) or the schedule disturbance caused when the user submits to multiple sites in order to guarantee the earliest possible start time.

4.2 Time-Sharing

The term time-sharing, or time-slicing, refers to the sharing of a processor's time among threads of different parallel programs. In such approaches, each processor executes a thread for some time, pauses it, and begins executing a new thread. Applications therefore exhibit short wait times but execute more slowly than under the dedicated set of processors provided by space-sharing Time-sharing is most often used to execute interactive jobs that do not necessarily

require peak performance. Interactivity is a critical requirement if parallel processing is to move beyond scientific supercomputing and into widespread deployment. Parallel processing on the desktop for example, is interactive. The same is true of web application infrastructures and consequently, many dynamic grid computing scenarios.

4.2.1 Local Scheduling

The simplest way to implement a parallel time-sharing scheduler is to run a uniprocessor system on each node and share a global run queue. Threads that are ready to execute are placed in the queue. When a processor becomes available, it simply removes the next thread from the queue, executes it for some time, and returns it to the back of the queue. This approach was once widely used in small-scale uniform memory access machines. An obvious advantage of this approach is fairness. Each thread receives an equal share of the machine and priority mechanisms are straightforward to enable. Local scheduling, however, is beset by numerous shortcomings. Contention for the global queue is a potential performance bottleneck, frequent context switching disturbs cache locality, and thread migration can be costly across processors, particularly when large chunks of data need be ported from one memory bank to another. The inefficiency of uncoordinated thread execution must be addressed by a more application-centric approach such as *gang scheduling*.

4.2.2 Gang Scheduling

The most accepted form of time-sharing is gang scheduling, an approach by which all threads of an application are executed concurrently as one *gang*. This approach is regarded as a union of space-sharing and time-sharing techniques and has been shown to outperform local scheduling in numerous studies.

Under gang scheduling, applications can perform more fine-grained communications without suffering a significant performance further, since gang scheduling assigns threads to processors, the approach enjoys all the benefits of affinity scheduling, including some local cache efficiencies and an obviated need for memory porting.

Gang scheduling, however, is limited in other respects. In addition to inheriting the drawbacks of memory management and context switching overheads from its time-sharing heritage, it also inherits many of the fragmentation issues of space-sharing several variations and relaxations of gang scheduling have been proposed to overcome this challenge.

One approach is *dynamic co scheduling*]. First proposed for commodity clusters, DCS observes that only threads that communicate often need be scheduled together and attempts to reduce fragmentation by scheduling each thread when a message arrives for it another inefficiency of gang scheduling, at least with respect to other time-sharing approaches, is its handling of I/O-intensive jobs. Such jobs cause degradation in both processor and I/O efficiency because gang scheduling fails to overlap I/O requests with computation. Processor efficiency suffers when threads idly await I/O results, neither making progress nor allowing compute intensive threads to execute

5. TRENDS IN PARALLEL PROCESSING

There are however, several trends that are reinvigorating the discipline and motivating novel scenarios and studies in parallel processing.

5.1 Parallelism in the Mainstream

Mainstream deployment of parallel applications may also drive innovation in programming models. OpenMP (Open Multi-Processing) for example, a shared-memory programming interface based on a fork-join model is making inroads deep enough to marginalize purely distributed memory architectures. OpenMP is not alone. The Department of Defense's High Productivity Computer Systems program constitutes an immense national effort to create the next generation of High Performance Computing (HPC) tools and architectures. All three industry partners (Sun, IBM, and Cray) are

developing new programming languages to accompany their architecture proposals. Independently, Microsoft is also developing a new version of C that eases the multi-threaded programming burden through implicit parallelism.

5.2 Grid Computing

A great source of new requirements for supercomputer scheduling is *grid computing*. Grid computing is the idea that a single community of users can gain access to multiple heterogeneous, physically distributed, and independently administered machines through a common interface. The purpose of grids is not necessarily to build a single, immensely powerful supercomputer, but rather to increase the utility of the machines involved through better load balancing and by exploiting application affinities Grid computing does not only create a new field of scheduling (grid scheduling), but also directly impacts the requirements of local schedulers. Because Grid computing systems must respect site autonomy, grid schedulers must be built to interface with each machine's local scheduler. In many respects today's local schedulers inadequately support effective grid scheduling. The most obvious obstacle is predictability. In order to decide which site is best for a particular job, a grid scheduler would have to determine the full response time of the job for each site. The unpredictability of queue wait times makes this very difficult. Even more important is the case when tasks need to be *co- allocated*, that is, scheduled to run at the same time but on different machines. This is a common requirement in work-flow based grid applications and is impossible to guarantee without support from the local scheduler. In response, some have proposed plan-based scheduling schemes in place of queuing approaches. Another feature of grid computing that cannot exist without local scheduler support is service level agreements. Such agreements can be rather arbitrary

Lastly, if local schedulers still require runtime estimates for each job, generating them automatically is unavoidable. When a user submits a job to a grid scheduler, he cannot be expected to even know which machines the job *may* run on, let alone provide a reasonably tight runtime estimate for every possibility. The estimate must be generated dynamically by the grid and local schedulers.

6. CONCLUSION

Job scheduling on parallel machines is a well studied research field that has led to widespread de facto standards: queued space-sharing with backfilling. This approach works well but can be improved through many techniques including automated runtime estimates, partial executions, and more intelligent processor allocation schemes.

Grid computing has created an entirely new field of scheduling research aimed at the efficient distribution of jobs across heterogeneous and independently administered machines. Concurrently, it is pressuring local scheduling research to provide expanded interfaces and re-evaluate scheduling objectives.

REFERENCE:

[1] Message Passing Interface Wikipedia free encyclopedia
[2] OpenMP website openmp.org
[3] 10th Workshop on Job Scheduling Strategies for Parallel Processing,
[4] D. G. Feitelson and L. R. L. Scheduling. Parallel Job Scheduling: Issues and Approaches.
[5] Grid computing : www.gridcomputing.com
[6] A. B. Downey. Using Queue Time Predictions for Processor Allocation

[7] Job Scheduling on Parallel Systems
Jonathan Weinberg University of
California,

IJSER