Dynamic Parallel Job Scheduling in Multi-cluster Computing Systems

J.H. Abawajy

Deakin University
School of Information Technology
Geelong, Victoria, Australia

Abstract. Job scheduling is a complex problem, yet it is fundamental to sustaining and improving the performance of parallel processing systems. In this paper, we address an on-line parallel job scheduling problem in heterogeneous multi-cluster computing systems. We propose a new space-sharing scheduling policy and show that it performs substantially better than the conventional policies.

1 Introduction

In the last few years, the trends in parallel processing system design and deployment have been moving away from a single powerful supercomputers to cooperative networked distributed systems such as commodity-based *cluster computing* systems. Research in cluster computing has focused on tools that are useful for putting together a cost-effective off-the-shelf high-performance cluster computing systems as well as developing application programs and executing them remotely. Aggregation of many resources is not enough to guarantee good performance - careful scheduling must be employed to achieve the best performance possible [6]. Without the support from well-designed cluster job scheduling policy, resources are shared in an ad-hoc manner, limiting performance as well as the utilization of the resources. Hence, one of the most important problems that must be addressed in order to realize the advantages of cluster computing systems is that of *job scheduling* problem.

Job scheduling problem has been extensively studied on parallel computers (e.g., [8], [4], [11], [2]) and to a lesser extent on cluster computing systems (e.g., [1] and [6]). Existing job scheduling policies can be classified into *space-sharing* (e.g., [4], [10], [11]) and *time-sharing* (e.g., [8]). It is also possible to combine these two types of policies into a *hybrid policy* as in [6], [2]. In a time-sharing policy, processors are shared over time by executing different applications on the same processors during different time intervals, which is commonly known as time-slice or quantum. In the space sharing approach, processors are partitioned into disjoint sets and each application executes in isolation on one of these sets.

In this paper, we focus on space-sharing policy. In general, based on when processor partition is created, space-sharing policies can be classified as *fixed*, *static*, and *dynamic* approaches. In fixed policy, processors are partitioned into

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 27–34, 2004.

a fixed size partitions at the time the system starts. The scheduler assigns one or more partitions to parallel jobs based on the size of the jobs. Most of the conventional cluster-based systems (e.g., LSF [1]) use fixed scheduling policy. The positive aspect of the fixed approach is its implementation simplicity. However, it does not adapt to changes both in the system load conditions and resource requirements of applications. Hence, it can lead to processor fragmentation problem [6], which in turn can lead to relatively low processor utilization and system throughput.

In static space sharing policies (e.g., [4], [11]), the partition size allocated to a job is determined at allocation time. Hence, it can adapt to the system load condition as such avoiding some problems associated with the fixed space-sharing approach. However, as in fixed approach, applications will hold the processors assigned to them until they terminate (i.e., for the lifetime of the application), which can also lead to processor fragmentation problem. The dynamic space-sharing approach eliminates most of the problems associated with fixed and static approaches. The basic idea behind the dynamic space-sharing policy is to make the jobs in the system share the processors equally as much as possible. This is achieved by varying the number of processors allocated to an application during its execution. This means processors may be reclaimed from a running job and distributed to newly arrived jobs, or additional processors may be added to an executing job when processors become available. Dynamic space-sharing policies are typically used in shared-memory systems since significant programming effort and execution overhead must be expended to change a job's processor allocation during execution. In a distributed memory environment especially, the costs of data repartitioning can overwhelm the scheduling benefits realized by malleable job support.

In this paper, we address the problem of scheduling parallel jobs in multiple cluster computing systems. Job scheduling is a challenging problem as the performance and applicability of the scheduling policy is highly sensitive to a number of factors such as machine architecture. In particular, cluster computing systems have several subtle but significant characteristics that influence and complicate scheduling decisions, which are not an issue in conventional parallel processing systems. Some of these factors include system heterogeneity, scale, interconnection technologies that typically exhibit high overhead and low bandwidth, and the availability of the system resources vary over time. This variation is both highly dynamic and unpredictable. Due to such inherent characteristics of cluster computing systems, scheduling strategies developed for the traditional distributed systems have to be extended significantly to support the dynamics of cluster computing systems. Dynamic scheduling is required since resources (i.e., machines and networks) may suffer dynamic load fluctuations or may be added or removed during the course of application execution.

The rest of the paper is organized as follows. In Section 2, the system model and the proposed scheduling policy are described. In Section 3, the performance evaluation of the proposed policy is described. It is shown that the proposed

policy performs substantially better than a baseline policy. Conclusion and future directions are described in Section 4.

2 Dynamic Scheduling Policy

2.1 System Model

For the most part, job scheduling research in cluster computing environments has focused on a single cluster computing systems (e.g, [9]) under a common assumption that all processors in the system have equal processing capacity (i.e., homogeneous) (e.g., [10], [9]) and dedicated (e.g., [10]). In contrast, we focus on *shared*, *heterogeneous*, and *multi-programmed* cluster computing systems composed (see Figure 1) of *multiple clusters* as in [6], [3] spanning campus wide area.

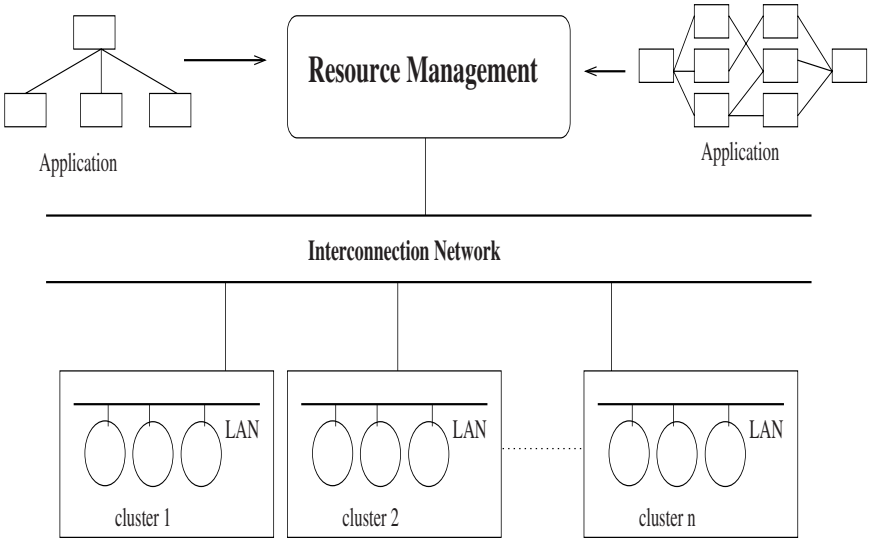

Fig. 1. Multicluster Computing System

We assume that a cluster is geographically compact and contains a set of workstations of similar architecture, connected by a fast Ethernet (100 Mbits/sec). Similarly, inter-cluster interconnection network is a fast Ethernet (100 Mbits/sec). We also assume that a network segment contains a single cluster only. Computers in different clusters do not share the communication bandwidth. Note that these assumptions can be easily relaxed, but allow us to ignore the mechanisms involved in getting jobs to run, and concentrate on policy issues for selecting which jobs as well as where and when they should run. In addition,

we restrict our focus to two dimensions of heterogeneity: processor speed and cluster sizes as these two are expected to be the major components of system heterogenity in networked-systems [6].

Adaptive space-sharing policies for cluster computing systems should have at least two components: *processor allocation* (i.e., how many processors is allocated to a job) and *processor selection* (i.e., which processors should be allocated to a job). However, the classical space-sharing policies (e.g., [2], [10]), consider only the processor allocation aspect while they are oblivious to processor selection problem. An approach to partition a moldable parallel job on distributed systems is described in [11] but not address the processor selection problem. Also, they are characterized with *all-or-nothing* approach, which means that jobs will hold on to the processors allocated to them even if they do not need it. Existing adaptive policies do not actively correct such a resource imbalance state. As a result, some clusters may be overloaded while others may sit idle. This leads to poor resource utilization due to *processor fragmentation*. The next section discusses the proposed scheduling policy.

2.2 Proposed Scheduling Policy

The proposed policy essentially mimics the conventional *dynamic space-sharing* approach such that idling processors are allocated to other jobs that can fruitfully utilize them. To achieve this, we keep a *pool of available processors* from which the scheduler allots a set of processors to unscheduled jobs. When a processor is assigned to a job completes the allocated task, it immediately returns to the pool where it becomes available for re-assignment to another job.

The activation of the scheduler occurs in three instances: (1) when a job arrives; (2) when a job departs; and (3) when there are α processors, $0 \leq \alpha \leq \mathbf{P}$, in the *pool*, where α is a tunable parameter and $\mathbf{P}$ is the total number of processors in the system. When the algorithm is invoked, it assigns a set of processors equal to the lesser of the partition size and the job's maximum parallelism. The target partition size is determined as follows:

$$target = min(\text{maximum parallelism of the job}, \text{partition size}) \tag{1}$$

where the maximum parallelism is given at the job arrival time and the *partition size* is computed as follows:

$$\text{partition size} = \left\lceil \frac{\mathbf{P}}{(\text{Queued Jobs}+1) + (0.5 \times \text{Executing Jobs})} \right\rceil \tag{2}$$

The scheduler then goes into processor selection phase, which selects appropriate processors for the job. Processor selection phase is performed as follows:

1. Select processors within the same cluster if possible
2. Select processors taking cluster proximity into account

Next, the scheduler determines the exact number of tasks assigned to each processor, P_i, in the target partition as follows:

$$bunch = ff(P_i) \times \frac{\text{Job Maximum Parallelism}}{ff(System)} \tag{3}$$

where parameter *ff* is called a fitness factor for each processors P_i in the system that is defined as follows:

$$ff = \frac{\text{CPU speed} \times \text{Default MPL}}{\text{slowest processor speed in the system}} \tag{4}$$

Finally, the tasks assigned to processor P_i are *folded* onto the target partition size allocated to the job. For example, if P_i is assigned 3 tasks, these tasks are folded into one large task assuming that jobs are malleable.

3 Performance Evaluation

We used discrete event simulation written in C programming language to study the performance of the proposed policy and compare it with the baseline policy [2]. For relative performance evaluation, we use the *mean response time (MRT)* and *average utilization*. The MRT is defined as the average job response time of completed jobs. The *average utilization* is defined as the job arrival rate times the mean service demand of the jobs divided by the number of processors in the system.

In this paper, we focus on moldable parallel workload. Moldable jobs are parallel jobs that are flexible in the number of processors at the time the job starts, but cannot be reconfigured during execution. The motivation for considering moldable parallel workload is that such workload constitutes a significant portion of the high-performance computing centers workloads and likely to increase in future. As in [2], we assume that jobs only request processors and we do not include in the model any other type of resources.

We used a system composed of 8 clusters, each cluster with 8 processors. In order to model the processors speed, we used the SPECfp2000 results for a family of Intel Pentium 3 and 4 processors with different clock speeds. We used 200μs for the context switch overhead, which is consistent with most of modern operating system for workstations. We set the space-sharing threshold ($\alpha = 2$), $MPL = 2$ and background job mean service demand=2.0. We model the communication overhead as follows:

$$T_{comm} = \text{Startup} + \frac{\text{Message size}}{\text{Bandwidth}} \tag{5}$$

The communication network latency to be 50μsec with the transfer rate of 100Mbits/sec.

The abstract model for parallel programs consists of a set of input parameters along with a job structure. Each job is characterized by *arrival time*, *service*

demand time in a dedicated environment, *maximum parallelism*, and the *size of the job* in Kbytes. Also, each can be decomposed into t tasks, $\mathbf{T}=\{T_1,...,T_t\}$ and each task T_i executes sequential code. The maximum parallelism of the jobs is uniformly distributed over the range of 1 to 64, while the service demands of the jobs are generated using hyper-exponential distribution with mean 14.06 [10], [2]. The default arrival CV is fixed at 1 (i.e., we assume Poisson arrivals) and the default service time CV is fixed at 3.5 as empirical observations at several supercomputer centers indicated this to be a reasonable value [10].

4 Results and Discussion

4.1 Relative Performance

We compared the relative performance of the proposed policy with the MPAP policy [2] in homogeneous environments. The result shows that at low system loads, there is no significant difference among the two policies as there are plenty of processors idle most of the time at this load level. However, as the load increases the dynamic policy performs better (by about 20% to 30%) than the baseline policy. This trend can be explained by the fact that in the MAP policy jobs tend to be allocated smaller partition-size as the system load increases and in the presence of local workload. As the number of processors allocated to a job decreases below its maximum parallelism, the service demand of the tasks also increases. This makes the jobs sensitive to the presence of the background load, which can increase the wait time of the jobs.

4.2 Sensitivity Analysis

We examined the impact of the background workload on the performance of the two policies. Note that the impact of the background load on the parallel job depends on the service demand of the parallel tasks; the longer a parallel task occupies the processor the more likely its execution is interrupted by the background workload. The result shows that the MAP policy is more sensitive than the dynamic policy. This is because the MAP policy suffers from a form of *processor fragmentation* induced by the presence of the background load and the way the partition-size is assigned to the jobs. Since partition size is computed based on the total number of processors in the system, the actual number of idle processors can be lower than the partition-size computed. This is because of the fact that some of the processors run background load at the time and the MAP policy does not take into account the background loads when computing the partition size. In this situation, the processors will remain idle as the scheduler will not assign them to jobs even if there are small jobs that can fruitfully utilize them.

Sensitivity of the two policies to the combined effects of background load and processor heterogeneity is also examined. At medium to high system loads, the MAP policy tends to be more sensitive than the dynamic policy. This is

due to the fact that, in addition to processor heterogeneity, the jobs experience slowdown due to interference from the background jobs. Note that heterogeneity can lead to a load imbalance situation. In such situations, the completion time of the jobs increases. When the execution of jobs in an already imbalanced environment is interrupted due to background jobs, the finishing time of the jobs is further increased.

In summary, the relatively poor performance of the MAP policy is due to the space sharing nature of the policy. Note that in the extreme case, the MAP policy ends up allocating a single processor to each job. Therefore, all the optimizations proposed to alleviate the problems associated with MAP policy have no impact at this stage. In addition, the presence of the background load also exacerbates the situation by creating interference for currently running jobs. This interference is observed as an increase in the completion time of the parallel jobs due to resource contention. The MAP policy reaches a point at which performance dramatically falls off. When the load is sufficiently high with all the processors running at least 1 job, the interference increases rapidly and performance can be very poor. Also under the MAP policy a job may have to be executed in one of the slower processors, considerably degrading the execution time.

5 Conclusion and Future Direction

In this paper, we investigated parallel job scheduling problem on heterogeneous networks of workstations, where computing power varies among the workstations, and local and parallel jobs may interact with each other in execution. We proposed a scheduling policy based on a virtually rooted tree structure that employs a pull-push scheme for scheduling and load balancing parallel jobs over multiple clusters. The proposed policy allows multiple job streams to share the system concurrently in an environment where the actual load distribution is not completely predictable in advance and scheduling is done dynamically. Also, the policy integrates several approaches (i.e. task scheduling, load balancing, self-scheduling, and time-space sharing) into a simple framework for parallel job scheduling on a system composed of multiple clusters. We studied the performance of the proposed scheduling policy through simulation. The results indicate that the proposed scheduling policy significantly better than the other scheduling policies used in the study.

Acknowledgments. Financial help is provided by Deakin University. The help of Maliha Omar is also greatly appreciated.

References

1. Ming Q. X.: Effective Metacomputing using LSF MultiCluster, Proceedings of CCGrid (2001) 100-106.
2. Thyagaraj, T.K. and Dandamudi, S. P.: An Efficient Adaptive Scheduling Scheme for Distributed Memory Multicomputers, IEEE Transactions on Parallel and Distributed Systems, **12**, (2001) 758-768.

3. Abawajy, J. H. and Dandamudi, S. P.: A Unified Resource Scheduling Approach on Cluster Computing Systems, Proceedings of the PDCS'03, (2003) 43-48.
4. Rosti, E. Smirni, E. Serazzi, G. and Dowdy, L. W.: Analysis of Non-Work-Conserving Processor Partitioning Policies, Proceedings of JSSPP (1995) 165-181.
5. Abawajy, J. H. and Dandamudi, S. P.: Scheduling Parallel Jobs with CPU and I/O Resource Requirements in Cluster Computing Systems, Proceedings of the 11th IEEE/ACM MASCOTS'03 (2003) 336-351.
6. Abawajy, J. H. and Dandamudi, S. P.: Parallel Job Scheduling on Multi-Cluster Computing Systems, Proceedings of the IEEE Cluster (2003) 11-17.
7. Feitelson, D. G. and Rudolph, L.: Toward Convergence in Job Schedulers for Parallel Supercomputers, Proceedings of JSSPP (1996) 1-26.
8. Feitelson, D. G. and Jette, M. A.: Improved Utilization and Responsiveness with Gang Scheduling", Proceedings of JSSPP (1997) 238-261.
9. Ryu, K.D. and Hollingsworth, J.K.: Exploiting Fine-Grained Idle Periods in Networks of Workstations, IEEE Transactions on Parallel and Distributed System, **11**, (2000) 683-698.
10. Stergios, A. V. and Sevcik, K. C.: Parallel Application Scheduling on Networks of Workstations, Journal of Parallel and Distributed Computing, **43** (1997) 1159-66.
11. Zhengao, Z. and Dandamudi,S. P.: An Adaptive Space-Sharing Policy for Heterogeneous Parallel Systems, HPCN'01 (2001) 353-362.