LNCS 4376

Eitan Frachtenberg
Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

12th International Workshop, JSSPP 2006
Saint-Malo, France, June 2006
Revised Selected Papers

Springer

Eitan Frachtenberg Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

12th International Workshop, JSSPP 2006
Saint-Malo, France, June 26, 2006
Revised Selected Papers

Springer

Preface

This volume contains the papers presented at the 12th workshop on Job Scheduling Strategies for Parallel Processing. The workshop was held in Saint-Malo, France, on June 16, 2006, in conjunction with SIGMETRICS 2006.

This year, the presented papers covered a large variety of topics. The first three papers address workflow problems. "Provably efficient two-level adaptive scheduling" by Yuxiong He et al. provides a theoretical analysis of a scheduling approach for independent jobs consisting of threads, that are represented by a DAG. Job and thread scheduling are separately addressed with different algorithms. The task graph is not known a priori in the paper "Scheduling dynamically spawned processes in MPI-2" by Márcia Cera et al., but processes are spawned dynamically. This paper is based on the features of MPI-2 and evaluates its scheduler with the help of an experiment. The DAG of a Grid job is known at submission time in the problem discussed in the paper "Advance reservation policies for workflows" by Henan Zhao and Rizos Sakellariou. Here, the tasks of this job are automatically scheduled on heterogeneous machines using advance reservation such that the overall execution time frame of the user is obeyed. The proposed approach is again experimentally evaluated.

The next three papers describe classical job scheduling problems that arise when parallel jobs are submitted to parallel systems with little or no node heterogeneity. The paper "On advantages of scheduling using Genetic Fuzzy systems" by Carsten Franke et al. presents scheduling algorithms that support arbitrary scheduling criteria. The algorithms are trained with recorded workloads using Fuzzy concepts. Their performances are evaluated by simulations with those workloads. In their paper "Moldable parallel job scheduling using job efficiency: An iterative approach," Gerald Sabin et al. show that scalability information of a job can help to improve the efficiency of this job. As in the previous paper, they use real workload traces for evaluation. The missing scalability information is provided with the help of a well-established speedup model. This model is also used in the paper "Adaptive job scheduling via predictive job resource allocation" by Lawrence Barsanti and Angela Sodan. Similar to the previous paper, the scalability of jobs improves the schedule performance. In addition, the resource allocation considers future job submissions based on a suitable prediction.

Many scientific applications are data intensive. For those applications, it is important to consider the network latency to transfer data from the storage facility to the parallel processing system. It is possible to improve schedule performance by scheduling those jobs on compute resources that are local to the storage resources. This is the subject of the paper "A data locality-aware online scheduling approach for I/O-intensive jobs with file sharing" by Gaurav Kanna et al. The next two papers address job migration issues. "Volunteer computing on clusters" by Deepti Vyas and Jaspal Subhlok demonstrates that nodes of a compute cluster are often underutilized while executing parallel applications. Exploiting this observation by a cycle stealing approach will lead only to a small slowdown of the parallel host application while system throughput increases significantly. Idleness of processors is also the subject of the paper "Load balancing: Toward the Infinite Network and Beyond" by Javier Bustos-Jiménez. There, active objects are sent to underutilized processors that are determined with the help of a peer-to-peer approach. The performance of the approach is evaluated by an experiment with a real application and also by simulations. Jonathan Weinberg and Allan Snavely observed in their paper "Symbiotic space-sharing on SDSC's DataStar system" that the hierarchical architecture of modern parallel processing systems leads to a significant amount of resource sharing among independent jobs and thus to performance degradation. They propose to generate better schedules by considering combinations of jobs with minimum interference between them. Again the performance is evaluated with the help of experiments with real applications.

The last two papers address job modeling issues in Grid computing. "Modeling job arrivals in a data-intensive Grid" by Hui Li et al. analyzes job arrival processes in workloads from high-energy physics and uses a special Markov process to model them. Virtual organizations determine the granularity of the model. The paper "On Grid performance evaluation using synthetic workloads" by Alexandru Iosup et al. discusses various aspects of performance analysis. The authors review different performance metrics and show important properties of existing workloads. Then, they present workload modeling requirements that are specific for Grid computing.

All submitted papers went through a complete review process, with the full version being read and evaluated by an average of five reviewers. We would like to thank the Program Committee members for their willingness to participate in this effort and their excellent, detailed reviews: Su-Hui Chiang, Walfredo Cirne, Allen Downey, Dror Feitelson, Allan Gottlieb, Andrew Grimshaw, Moe Jette, Richard Lagerstrom, Virginia Lo, Jose Moreira, Bill Nitzberg, Mark Squillante, John Towns, Jon Weissman, and Ramin Yahyapour.

The continued interest in this area is reflected by the longevity of this workshop, which has now reached its 12th consecutive year. The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, 2862, 3277, and 3834 (and since 1998 they have also been available online).

Finally, we would like to give our warmest thanks to Dror Feitelson and Larry Rudolph, the founding co-organizers of the workshop. Their efforts to promote this field are evidenced by the continuing success of this workshop.

November 2006

Eitan Frachtenberg
Uwe Schwiegelshohn

Table of Contents

Provably Efficient Two-Level Adaptive Scheduling*

Yuxiong He[1], Wen-Jing Hsu[1], and Charles E. Leiserson[2]

[1] Nanyang Technological University, Nanyang Avenue 639798, Singapore
yxhe@mit.edu, hsu@ntu.edu.sg
[2] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
cel@mit.edu

Abstract. Multiprocessor scheduling in a shared multiprogramming environment can be structured in two levels, where a kernel-level job scheduler allots processors to jobs and a user-level thread scheduler maps the ready threads of a job onto the allotted processors. This paper presents two-level scheduling schemes for scheduling "adaptive" multithreaded jobs whose parallelism can change during execution. The AGDEQ algorithm uses dynamic-equipartioning (DEQ) as a job-scheduling policy and an adaptive greedy algorithm (A-GREEDY) as the thread scheduler. The ASDEQ algorithm uses DEQ for job scheduling and an adaptive work-stealing algorithm (A-STEAL) as the thread scheduler. AGDEQ is suitable for scheduling in centralized scheduling environments, and ASDEQ is suitable for more decentralized settings. Both two-level schedulers achieve $O(1)$-competitiveness with respect to makespan for any set of multithreaded jobs with arbitrary release time. They are also $O(1)$-competitive for any batched jobs with respect to mean response time. Moreover, because the length of the scheduling quantum can be adjusted to amortize the cost of context-switching during processor reallocation, our schedulers provide control over the scheduling overhead and ensure effective utilization of processors.

1 Introduction

Multiprocessors are often used for multiprogrammed workloads where many parallel applications share the same machine. As Feitelson points out in his excellent survey [27], schedulers for these machines can be implemented using two levels: a kernel-level ***job scheduler*** which allots processors to jobs, and a user-level ***thread scheduler*** which maps the threads belonging to a given job onto the allotted processors. The job schedulers may implement either ***space-sharing***, where jobs occupy disjoint processor resources, or ***time-sharing***, where different jobs may share the same processor resources at different times. Moreover, both the thread scheduler and the job scheduler may be either ***adaptive*** (called "dynamic" in [19]), allowing the number of processors allotted to a job to vary

* This research was supported in part by the Singapore-MIT Alliance and NSF Grants ACI-0324974 and CNS-0540248.

while the job is running, or ***nonadaptive*** (called "static" in [19]), where a job runs on a fixed number of processors over its lifetime. A ***clairvoyant*** scheduling algorithm may use knowledge of the jobs' execution time, whereas a ***non-clairvoyant*** algorithm assumes nothing about the execution time of the jobs. This paper presents two provably efficient two-level adaptive schedulers, each of which schedules jobs nonpreemptively and without clairvoyance.

With ***adaptive*** scheduling [4] (called "dynamic" scheduling in many other papers [27,60,41,58,37]), the job scheduler can change the number of processors allotted to a job while the job executes. Thus, new jobs can enter the system, because the job scheduler can simply recruit processors from the already executing jobs and allot them to the new jobs. Without an adequate feedback mechanism, however, both adaptive and nonadaptive schedulers may waste processor cycles, because a job with low parallelism may be allotted more processors than it can productively use.

If individual jobs provide ***parallelism feedback*** to the job scheduler, waste can be avoided. When a job does not require many processors, it can release the excess processors to the job scheduler to be reallotted to jobs in need. When a job needs more processors, it can make a request to the job scheduler. Based on this parallelism feedback, the job scheduler can adaptively change the allotment of processors according to the availability of processors and the system administrative policy.

A two-level scheduler communicates the parallelism feedback by each job requesting processors from a job scheduler at regular intervals, called ***quanta***. The quantum length is typically chosen to be long enough to amortize the scheduling overheads, including the cost of reallotting processors among the jobs. The job scheduler uses the parallelism feedback to assign the available processors to the jobs according to its administrative policy. During the quantum, the job's allotment does not typically change. Once a job is allotted processors, the job's thread scheduler maps the job's threads onto the allotted processors, reallocating them if necessary as threads are spawned and terminated.

Various researchers [21,20,29,41,59] have proposed the use of ***instantaneous parallelism*** — the number of processors the job can effectively use at the current moment — as the parallelism feedback to the job scheduler. Unfortunately, using instantaneous parallelism as feedback can either cause gross misallocation of processor resources [49] or introduce significant scheduling overhead. For example, the parallelism of a job may change substantially during a scheduling quantum, alternating between parallel and serial phases. Depending on which phase is currently active, the sampling of instantaneous parallelism may lead the task scheduler to request either too many or too few processors. Consequently, the job may either waste processor cycles or take too long to complete. On the other hand, if the quantum length is set to be small enough to capture frequent changes in instantaneous parallelism, the proportion of time spent reallotting processors among the jobs increases, resulting in a high scheduling overhead.

A-GREEDY [1] and A-STEAL [2,3] are two adaptive thread schedulers that provide the parallelism feedback to the job scheduler. Rather than using instantaneous parallelism, these thread schedulers employ a single summary statistic and the job's behavior in the previous quantum to make processor requests of the job scheduler. Even though this parallelism feedback is generated based on the job's history and may not be correlated to the job's future parallelism, A-GREEDY and A-STEAL still guarantee to make effective use of the available processors.

Intuitively, if each job provides good parallelism feedback and makes productive use of available processors, a good job scheduler should ensure that *all* the jobs perform well. In this paper, we affirm this intuition for A-GREEDY and A-STEAL in the case when the job scheduler implements dynamic equipartitioning (DEQ) [55, 41]. DEQ gives each job a fair allotment of processors based on the job's request, while allowing processors that cannot be used by a job to be reallocated. DEQ was introduced by McCann, Vaswani, and Zahorjan [41] based on earlier work on equipartitioning by Tucker and Gupta [55], and it has been studied extensively [21, 20, 29, 42, 41, 24, 36, 46, 45, 59, 40, 25].

This paper shows that efficient two-level adaptive schedulers can ensure that all jobs can perform well. AGDEQ, which couples DEQ with A-GREEDY, is suitable for centralized thread scheduling, such as might be used to schedule data-parallel jobs, wherein each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner. ASDEQ, which couples DEQ with A-STEAL, is suitable when each job distributes threads over the allotted processors using decentralized work-stealing [16, 31, 47, 13].

The main contributions of this paper are as follows. In a centralized environment, AGDEQ guarantees $O(1)$-competitiveness against an optimal clairvoyant scheduler with respect to makespan. For any set of batched jobs, where all jobs have the same release time, AGDEQ also achieves $O(1)$-competitiveness with respect to mean response time. In a decentralized settings where the scheduler has no knowledge of all the available threads at the current moment, ASDEQ guarantees $O(1)$-competitiveness with respect to makespan for any set of jobs with arbitrary job release time. It is also $O(1)$-competitive with respect to the mean response time for batched jobs. Unlike many previous results, which either assume clairvoyance [38, 18, 43, 33, 34, 56, 48, 50, 57] or use instantaneous parallelism [21, 14, 22], our schedulers remove these restrictive assumptions. We generate parallelism feedback after each quantum based on the job's behavior in the past quantum. Even though job's future parallelism may not be correlated with its history of parallelism, our schedulers can still guarantee constant competitiveness for both the makespan and the mean response time. Moreover, because the quantum length can be adjusted to amortize the cost of context-switching during processor reallocation, our schedulers provide control over the scheduling overhead and ensure effective utilization of processors.

The remainder of this paper is organized as follows. Section 2 describes the job model, scheduling model, and objective functions. Section 3 describes the AGDEQ algorithm. Section 4 and 5 analyze the competitiveness of AGDEQ with respect to makespan and mean response time, respectively. Section 6 presents the ASDEQ algorithm and analyzes its performance. Section 7 gives a lower bound on the competitiveness for mean response time. Section 9 concludes the paper by raising issues for future research.

2 Models and Objective Functions

This section provides the background formalisms for two-level scheduling, which will be used to study AGDEQ and ASDEQ. We formalize the job model, define the scheduling model, and present the optimization criteria of makespan and mean response time.

Job Model
A ***two-level scheduling problem*** consists of a collection of independent jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_{|\mathcal{J}|}\}$ to be scheduled on a collection of P identical processors. This paper restricts its attention to the situation where $|\mathcal{J}| \leq P$, that is, the number of jobs does not exceed the number of processors. (The situation where the parallel computer may sometimes be heavily loaded with jobs remains an interesting open problem.) Like prior work on scheduling of multithreaded jobs [12,13,11,10,8,26,32,44], we model the execution of a multithreaded job J_i as a dynamically unfolding directed acyclic graph (dag) such that $J_i = (V(J_i), E(J_i))$ where $V(J_i)$ and $E(J_i)$ represent the sets of J_i's vertices and edges, respectively. Similarly, let $V(\mathcal{J}) = \bigcup_{J_i \in \mathcal{J}} V(J_i)$. Each vertex $v \in V(\mathcal{J})$ represents a unit-time instruction. The ***work*** $T_1(i)$ of the job J_i corresponds to the total number of vertices in the dag, that is, $T_1(i) = |V(J_i)|$. Each edge $(u, v) \in E(J_i)$ represents a dependency between the two vertices. The precedence relationship $u \prec v$ holds if and only if there exists a path from vertex u to vertex v in $E(J_i)$. The ***critical-path length*** $T_\infty(i)$ corresponds to the length of the longest chain of precedence dependencies. The ***release time*** $r(i)$ of the job J_i is the time immediately after which J_i becomes first available for processing. For a ***batched*** job set $\mathcal{J}$, all jobs in $\mathcal{J}$ have the same release time. (Without loss of generality, we assume that $r(i) = 0$ for all $J_i \in \mathcal{J}$.)

Scheduling Model
Our scheduling model assumes that time is broken into a sequence of equal-sized ***scheduling quanta*** $1, 2, \ldots$, each of length L, where each quantum q includes the interval $[Lq, Lq+1, \ldots, L(q+1)-1]$ of time steps. The quantum length L is a system configuration parameter chosen to be long enough to amortize scheduling overheads. These overheads might include the time to reallocate processors among the various jobs and the time for the thread scheduler to communicate with the job scheduler, which typically involves a system call.

The job scheduler and thread schedulers interact as follows. The job scheduler may reallocate processors between quanta. Between quantum $q-1$ and quantum q, the thread scheduler (for example, A-GREEDY or A-STEAL) of a given job J_i determines the job's ***desire*** $d(i, q)$, which is the number of processors J_i wants for quantum q. The thread scheduler provides the desire $d(i, q)$ to the job scheduler as its parallelism feedback. Based on the desire of all running jobs, the job scheduler follows its processor-allocation policy (for example, dynamic equi-partitioning) to determine the ***allotment*** $a(i,q)$ of the job with the constraint that $a(i,q) \leq d(i,q)$. Once a job is allotted its processors, the allotment does not change during the quantum. Consequently, the thread scheduler must do a good job in estimating how many processors it will need in the next quantum, as well as scheduling the ready threads on the allotted processors. Moreover, the thread scheduler must operate in an online and nonclairvoyant manner, oblivious to the future characteristics of the dynamically unfolding dag.

A ***schedule*** $\chi = (\tau, \pi)$ of a job set $\mathcal{J}$ on P processors is defined as two mappings $\tau : V(\mathcal{J}) \rightarrow \{1, 2, \ldots, \infty\}$ and $\pi : V(\mathcal{J}) \rightarrow \{1, 2, \ldots, P\}$, which map the vertices in the job set $\mathcal{J}$ to the set of time steps and to the set of processors in the machine, respectively. A valid mapping must preserve the precedence relationship of each job: for any two vertices $u, v \in V(\mathcal{J})$, if $u \prec v$, then $\tau(u) < \tau(v)$, that is, the vertex u must be executed before the vertex v. A valid mapping must also ensure that a processor is only assigned to one job at any time: for any two distinct vertices $u, v \in V(\mathcal{J})$, we have $\tau(u) \neq \tau(v)$ or $\pi(u) \neq \pi(v)$.

Objective Functions

We can now define the objective functions that a two-level scheduler should minimize.

Definition 1. *Let χ be a schedule of a job set $\mathcal{J}$ on P processors. The **completion time** a job $J_i \in \mathcal{J}$ is*

$$T_\chi(i) = \max_{v \in V_i} \tau(v) ,$$

*and the **makespan** of $\mathcal{J}$ is*

$$\mathrm{T}_\chi(\mathcal{J}) = \max_{J_i \in \mathcal{J}} T_\chi(i) .$$

*The **response time** of a job $J_i \in \mathcal{J}$ is*

$$\mathrm{R}_\chi(i) = T_\chi(i) - r(i) ,$$

*the **total response time** of $\mathcal{J}$ is*

$$\mathrm{R}_\chi(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} \mathrm{R}_\chi(i) ,$$

*and the **mean response time** of $\mathcal{J}$ is*

$$\overline{\mathrm{R}}_\chi(\mathcal{J}) = \mathrm{R}_\chi(\mathcal{J}) / |\mathcal{J}| .$$

That is, the completion time of J_i is simply the time at which the schedule completes the execution of J_i. The makespan of $\mathcal{J}$ is the time taken to complete all jobs in the job set. The response time of a job J_i is the duration between its release time $r(i)$ and the completion time $T_\chi(i)$. The total response time of a job set is the sum of the response times of the individual jobs, and the mean response time is the arithmetic average of the jobs' response times. For batched jobs where $r(i) = 0$ for all $J_i \in \mathcal{J}$, the total response time simplifies to $\mathrm{R}_\chi(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_\chi(i)$.

Competitiveness
The competitive analysis of an online scheduling algorithm compares the algorithm against an optimal clairvoyant algorithm. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan of the jobset $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler, and $\chi(A)$ denote the schedule produced by an algorithm A for the job set $\mathcal{J}$. A deterministic algorithm A is said to be ***c-competitive*** if there exist constants $c > 0$ and $b \geq 0$ such that $\mathrm{T}_{\chi(A)}(\mathcal{J}) \leq c \cdot \mathrm{T}^*(\mathcal{J}) + b$ holds for the schedule $\chi(A)$ of each job set. A randomized algorithm A is said to be ***c-competitive*** if there exists constants $c > 0$ and $b \geq 0$ such that $\mathrm{E}\left[\mathrm{T}_{\chi(A)}(\mathcal{J})\right] \leq c \cdot \mathrm{T}^*(\mathcal{J}) + b$ holds for the schedule $\chi(A)$ of each job set. Thus, for each job set $\mathcal{J}$, a c-competitive algorithm is guaranteed to have makespan (or expected makespan) within a factor c of that incurred in the optimal clairvoyant algorithm (up to the additive constant b). We shall show that AGDEQ and ASDEQ are c-competitive with respect to makespan, where $c > 0$ is a small constant. For the mean response time, we shall show that our algorithm is $O(1)$-competitive for batched jobs.

3 The AGDEQ Algorithm

AGDEQ is a two-level adaptive scheduler, which uses A-Greedy [1] as its thread scheduler and DEQ [41] as its job scheduler. Given a set $\mathcal{J}$ of jobs and P processors, DEQ works at the kernel level, partitioning the P processors among the jobs. Within each job, A-Greedy schedules threads at user level onto the allotted processors. The interactions between DEQ and A-Greedy follow the scheduling model described in Section 2. At the beginning of each quantum q, the A-Greedy thread scheduler for each job $J_i \in \mathcal{J}$ provides its desire $d(i, q)$ as parallelism feedback to the DEQ job scheduler. DEQ collects the desire information from all jobs and decides the allotment $a(i, q)$ for each job J_i. In this section, we briefly overview the basic properties of A-Greedy and DEQ.

The Adaptive Greedy Thread Scheduler
A-Greedy [1] is an adaptive greedy thread scheduler with parallelism feedback. In a two-level adaptive scheduling system, A-Greedy performs the following functions.

- Between quanta, it estimates its job's desire and requests processors from the job scheduler using its ***desire-estimation algorithm***.
- During the quantum, it schedules the ready threads of the job onto the allotted processors using its ***thread-scheduling algorithm***.

We now describe each of these algorithms.

A-Greedy's desire-estimation algorithm is parameterized in terms of a ***utilization parameter*** $\delta > 0$ and a ***responsiveness parameter*** $\rho > 1$, both of which can be tuned to affect variations in guaranteed bounds for waste and completion time.

Before each quantum, A-GREEDY for a job $J_i \in \mathcal{J}$ provides parallelism feedback to the job scheduler based on the J_i's history of utilization for the previous quantum. A-GREEDY classifies quanta as "satisfied" versus "deprived" and "efficient" versus "inefficient." A quantum q is ***satisfied*** if $a(i,q) = d(i,q)$, in which case J_i's allotment is equal to its desire. Otherwise, the quantum is ***deprived***. The quantum q is ***efficient*** if A-GREEDY utilizes no less than a δ fraction of the total allotted processor cycles during the quantum, where δ is the utilization parameter. Otherwise, the quantum is ***inefficient***. Of the four possibilities of classification, however, A-GREEDY only uses three: inefficient, efficient-and-satisfied, and efficient-and-deprived.

Using this three-way classification and the job's desire for the previous quantum, A-GREEDY computes the desire for the next quantum using a simple multiplicative-increase, multiplicative-decrease strategy. If quantum $q-1$ was inefficient, A-GREEDY decreases the desire, setting $d(i,q) = d(i,q-1)/\rho$, where ρ is the responsiveness parameter. If quantum $q-1$ was efficient and satisfied, A-GREEDY increases the desire, setting $d(i,q) = \rho d(i,q-1)$. If quantum $q-1$ was efficient but deprived, A-GREEDY keeps desire unchanged, setting $d(i,q) = d(i,q-1)$.

A-GREEDY's thread-scheduling algorithm is based on greedy scheduling [28, 15,12]. After A-GREEDY for a job $J_i \in \mathcal{J}$ receives its allotment $a(i,q)$ of processors from the job scheduler, it simply attempts to keep the allotted processors as busy as possible. During each time step, if there are more than $a(i,q)$ ready threads, A-GREEDY schedules any $a(i,q)$ of them. Otherwise, it schedules all of them.

The Dynamic-Equipartitioning Job Scheduler

DEQ is a dynamic-equipartitioning job scheduler [55,41] which attempts to give each job a fair share of processors. If a job cannot use its fair share, however, DEQ distributes the extra processors across the other jobs. More precisely, upon receiving the desires $\{d(i,q)\}$ from the thread schedulers of all jobs $J_i \in \mathcal{J}$, DEQ executes the following ***processor-allocation algorithm***:

1. Set $n = |\mathcal{J}|$. If $n = 0$, return.
2. If the desire for every job $J_i \in \mathcal{J}$ satisfies $d(i,q) \geq P/n$, assign each job $a(i,q) = P/n$ processors.
3. Otherwise, let $\mathcal{J}' = \{J_i \in \mathcal{J} : d(i,q) < P/n\}$. Allot $a(i,q) = d(i,q)$ processors to each $J_i \in \mathcal{J}'$. Update $\mathcal{J} = \mathcal{J} - \mathcal{J}'$. Go to Step 1.

Accordingly, for a given quantum all jobs receive the same number of processors to within 1, unless their desire is less. To simplify the analysis in this paper, we shall assume that all deprived jobs receive exactly the same number of processors, which we term the ***mean deprived allotment*** for the quantum. Relaxing this assumption may double the execution-time bound of a job, but our algorithms remain $O(1)$-competitive. A tighter but messier analysis retains the constants of the simpler analysis presented here.

4 Makespan of AGDEQ

This section shows that AGDEQ is c-competitive with respect to makespan for a constant $c \geq 1$. The exact value of c is related to the choice of the utilization parameter and responsiveness parameter in A-GREEDY. In this section, we first review lower bounds for makespan. Then, we analyze the competitiveness of AGDEQ in the simple case where all jobs are released at time step 0 and the scheduling quantum length is $L = 1$. Finally, we analyze the competitiveness of AGDEQ for the general case.

Lower Bounds

Given a job set $\mathcal{J}$ and P processors, lower bounds on the makespan of any job scheduler can be obtained based on release time, work, and critical-path length. Recall that for a job $J_i \in \mathcal{J}$, the quantities $r(i)$, $T_1(i)$, and $T_\infty(i)$ represent the release time, work, and critical-path length of J_i, respectively. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan produced by an optimal scheduler on a job set $\mathcal{J}$ scheduled on P processors. Let $T_1(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_1(i)$ denote the total work of the job set. The following two inequalities give two lower bounds on the makespan [14]:

$$\mathrm{T}^*(\mathcal{J}) \geq \max_{J_i \in \mathcal{J}} \{r(i) + T_\infty(i)\} \ , \tag{1}$$

$$\mathrm{T}^*(\mathcal{J}) \geq T_1(\mathcal{J})/P \ . \tag{2}$$

Analysis of a Simple Case

To ease the understanding of the analysis, we first consider the simple case where all jobs are released at time step 0 and the quantum length $L = 1$. We show that in this case, AGDEQ is $O(1)$-competitive with respect to makespan. Afterward, we shall extend the analysis to the general case.

The next two lemmas, proved in [1], bound the satisfied steps and the waste of any single job scheduled by A-GREEDY when the quantum length is $L = 1$. We restate them as a starting point for our analysis.

Lemma 1. *[1] Suppose that* A-GREEDY *schedules a job J_i with critical-path length $T_\infty(i)$ on a machine with P processors. Let $\rho = 2$ denote* A-GREEDY*'s responsiveness parameter, $\delta = 1$ its utilization parameter, and $L = 1$ the quantum length. Then,* A-GREEDY *produces at most $2T_\infty(i) + \lg P + 1$ satisfied steps.* □

Lemma 2. *[1] Suppose that* A-GREEDY *schedules a job J_i with work $T_1(i)$ on a machine. If $\rho = 2$ is* A-GREEDY*'s responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then* A-GREEDY *wastes no more than $2T_1(i)$ processor cycles in the course of the computation.* □

The next lemma shows that for the simple case, AGDEQ is $O(1)$-competitive with respect to makespan. Let $\chi = (\tau, \pi)$ be the schedule of a job set $\mathcal{J}$ produced by AGDEQ. For simplicity we shall use the notation $\mathrm{T}(\mathcal{J}) = \mathrm{T}_\chi(\mathcal{J})$ for the remaining of the section.

Lemma 3. *Suppose that a job set $\mathcal{J}$ is scheduled by* AGDEQ *on a machine with P processors, and suppose that all jobs arrive at time* 0. *Let $\rho = 2$ denote* A-GREEDY*'s responsiveness parameter, $\delta = 1$ its utilization parameter, and L the quantum length. Then, the makespan of $\mathcal{J}$ is bounded by*

$$\mathrm{T}(\mathcal{J}) \le 5\mathrm{T}^*(\mathcal{J}) + \lg P + 1 \ ,$$

where $\mathrm{T}^*(\mathcal{J})$ is the makespan produced by an optimal clairvoyant scheduler.

Proof. Suppose that the job J_k is the last job completed in the execution of the job set $\mathcal{J}$ scheduled by AGDEQ. Since the scheduling quantum length is $L = 1$, we can treat each scheduling quantum as a time step. Let $S(k)$ and $D(k)$ denote the set of satisfied steps and the set of deprived steps respectively for job J_k. Since J_k is the last job completed in the job set, we have $\mathrm{T}(\mathcal{J}) = |S(k)| + |D(k)|$. We bound $|S(k)|$ and $|D(k)|$ separately.

By Lemma 1, we know that the number of satisfied steps for job J_k is $|S(k)| \le 2T_\infty(i) + \lg P + 1$.

We now bound the number of deprived steps for J_k. If a step t is deprived for job J_k, the job gets fewer processors than it requested. On such a step $t \in D(k)$, DEQ must have allotted all the processors, and so we have $\sum_{J_i \in \mathcal{J}} a\,(i,t) = P$, where $a\,(i,t)$ denotes the allotment of the job J_i on step t. Let $a\,(\mathcal{J}, D(k)) = \sum_{t \in D(k)} \sum_{J_i \in \mathcal{J}} a\,(i,t)$ denote the total processor allotment of all jobs in $\mathcal{J}$ over J_k's deprived steps $D(k)$. We have $a\,(\mathcal{J}, D(k)) = \sum_{t \in D(k)} \sum_{J_i \in \mathcal{J}} a\,(i,t) = \sum_{t \in D(k)} P = P\,|D(k)|$. Since any allotted processor is either working on the ready threads of the job or wasted because of insufficient parallelism, the total allotment for any job J_i is bounded by the sum of its total work $T_1(i)$ and its total waste $w(i)$. By Lemma 2, the waste for the job J_i is $w(i) \le 2T_1(i)$, which is at most twice its work. Thus, the total allotment for job J_i is at most $3T_1(i)$, and the total allotment for all jobs is at most $\sum_{J_i \in \mathcal{J}} 3T_1(i) = 3T_1(\mathcal{J})$. Therefore, we have $a\,(\mathcal{J}, D(k)) \le 3T_1(\mathcal{J})$. Given that $a\,(\mathcal{J}, D(k)) \le 3T_1(\mathcal{J})$ and $a\,(\mathcal{J}, D(k)) = P\,|D(k)|$, we have $|D(k)| \le 3T_1(\mathcal{J})/P$.

Thus, we have $\mathrm{T}(\mathcal{J}) = |S(k)| + |D(k)| \le 3T_1(\mathcal{J})/P + 2T_\infty(k) + \lg P + 1$. Combining this bound with Inequalities (1) and (2), we obtain $\mathrm{T}(\mathcal{J}) \le 5\mathrm{T}^*(\mathcal{J}) + \lg P + 1$.

Since P is the number of processors on the machine, which is an independent variable with respect to any job set $\mathcal{J}$, Lemma 3 indicates that AGDEQ is 5-competitive with respect to makespan.

Analysis of the General Case

With the intuition from the simple case in hand, we now generalize the makespan analysis of AGDEQ to job sets with arbitrary job release times and scheduled with any quantum length L. First, we state two lemmas from [1] that describe the satisfied steps and the waste of a single job scheduled by A-GREEDY. Then, we show that AGDEQ is $O(1)$-competitive with respect to makespan in the general case.