Brief Announcement: Scheduling Parallelizable Jobs Online to Maximize Throughput

Kunal Agrawal, Jing Li, Kefu Lu, Benjamin Moseley
Washington University in St. Louis
1 Brookings Drive, St. Louis, MO 63130
{kunal,li.jing,kefulu,bmoseley}@wustl.edu

ABSTRACT

We consider scheduling parallelizable jobs online to maximize the throughput or profit of the schedule. A set of n jobs arrive online and each job J_i has an associated function $p_i(t)$, the profit obtained for finishing job J_i at time t. Each job has its own arbitrary non-increasing profit function. We consider the case where each job is a parallel job that can be represented as a directed acyclic graph (DAG). We give the first non-trivial results for the profit scheduling problem for DAG jobs showing $O(1)$-competitive algorithms using resource augmentation.

1 INTRODUCTION

Scheduling preemptive jobs online to meet deadlines is a fundamental problem and, consequently, this area has been extensively studied. In a typical setting, there are n jobs that arrive over time. Each job J_i arrives at time r_i, has a deadline d_i, relative deadline $D_i = d_i - r_i$ and a profit or weight p_i that is obtained if the job is completed by its deadline. The ***throughput*** of a schedule is the total profit of the jobs completed by their deadlines, and a popular scheduling objective is to maximize the throughput of the schedule.

In a generalization of the throughput problem, each job J_i is associated with a function $p_i(t)$, which specifies the profit obtained for finishing job J_i at time $r_i + t$. It is assumed that p_i can be different for each job J_i and the functions are arbitrary non-increasing functions. We call this problem the ***general profit*** problem.

In this work, we consider the throughput and general profit scheduling problems in the preemptive online setting for parallel jobs. In this setting, the *online* scheduler is only aware of the job at the time it arrives in the system, and a job is *preemptive* if it can be started, stopped, and resumed from the previous position later. We model each parallel job as a ***directed acyclic graph (DAG)***. Each node in the DAG is a sequence of instructions to be executed; the edges in the DAG represent dependencies. A node can be executed if and only if all of its predecessors have been completed. Therefore, two nodes can potentially run in parallel if neither precedes the other in the DAG. In this setting, each job J_i arrives as a single independent DAG and a profit of p_i is obtained if *all* nodes of the DAG are completed by job J_i's deadline. The DAG model can represent parallel programs written in many widely used parallel languages and libraries, such as OpenMP [25], Cilk Plus [16], Intel TBB [27] and Microsoft Parallel Programming Library [13].

Both the throughput and general profit scheduling problem have been studied extensively for sequential jobs. In the simplest setting, each job J_i has work or processing time W_i to be processed on a single machine. It is known that there exists a deterministic algorithm which is $O(\delta)$-competitive, where δ is the ratio of the maximum to minimum density of a job [10, 11, 20, 32]. The *density* of job J_i is $\frac{p_i}{W_i}$ (the ratio of its profit to its work). In addition, this is the best possible result for any deterministic online algorithm even in the case where all jobs have unit profit and the goal is to complete as many jobs as possible by their deadlines. If the algorithm can be randomized, $\Theta(\min\{\log\delta, \log\Delta\})$ is the optimal competitive ratio [17, 19]. Δ is the ratio of the maximum to minimum job processing time.

These strong lower bounds on the competitive ratio of any online algorithm makes it difficult to differentiate between algorithms and to discover the key algorithmic ideas that work well in practice. To overcome this challenge, the now standard form of analysis in scheduling theory is a *resource augmentation* analysis [18, 30]. In a resource augmentation analysis, the algorithm is given extra resources over the adversary and the competitive ratio is bounded. A s-speed c-competitive algorithm is given a processor that is s times faster than the optimal solution and achieves a competitive ratio of c. The seminal scheduling paper [18] considered the throughput scheduling problem and gave the best possible $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm for any fixed $\epsilon > 0$.

Since this work, there has been an effort to understand and develop algorithms for more general scheduling environments and objectives. In the identical machine setting where the jobs can be scheduled on m identical parallel machines, a $(1+\epsilon)$-speed $O(1)$-competitive algorithm is known for fixed $\epsilon > 0$ [4]. This has been extended to the case where the machines have speed scalable processors and the scheduler is energy aware [26]. In the related machines and unrelated machines settings, similar results have been obtained as well [15]. In [23] similar results were obtained in a distributed model.

None of the prior work considers parallel jobs. Parallel DAG jobs have been widely considered in scheduling theory for other objectives [1–3, 14, 22, 24, 28, 29]. There has been an extensive study in the real-time system community on how to schedule parallel DAG jobs by their deadlines [5–9, 12, 21, 22, 29]. These works focus on determining if a set of reoccurring jobs can *all* be completed by their deadlines, in contrast to optimizing throughput or profit.

Results: We give the *first* non-trivial results for scheduling parallelizable DAG jobs online to maximize throughput and then we

SPAA'17, July 24–26, 2017, Washington, DC, USA

ACM ISBN 978-1-4503-4593-4/17/07…$15.00
http://dx.doi.org/10.1145/3087556.3087590

generalize these results to the general profit problem. Two important parameters in the DAG setting are the ***critical-path length*** L_i of job J_i (its execution time on an infinite number of processors) and its total ***work*** W_i (its uninterrupted execution time on a single processor). The value of $\max\{L_i, W_i/m\}$ is a lower bound on the amount of time any 1-speed scheduler takes to complete job J_i on m cores. We will focus on schedulers that are aware of the values of L_i and W_i when the job arrives, but are unaware of the internal structure of the job's DAG. That is, besides L_i and W_i, the only other information a scheduler has on a job's DAG is which nodes are currently available to execute. We call such an algorithm *semi-non-clairvoyant* — for DAG jobs. This is a reasonable model for the real world programs since the DAG generally unfolds dynamically as the program executes. We first state a simple theorem about these schedulers.

Theorem 1.1. *There exists inputs where any semi-non-clairvoyant scheduler requires a speed augmentation of $2 - 1/m$ to be $O(1)$-competitive for maximizing throughput.*

Scheduling even a single DAG job in time smaller than $\frac{W_i - L_i}{m} + L_i$ is a hard problem even in the offline setting where the entire job structure is known in advance. This is captured by the classic problem of scheduling a precedence constrained jobs to minimize the makespan. For this problem, there is no $2 - \epsilon$ approximation assuming a variant of the unique games conjecture [31]. We can construct a DAG where any semi-non-clairvoyant scheduler will take roughly $\frac{W_i - L_i}{m} + L_i$ time to complete, while a fully clairvoyant scheduler can finish in time W_i/m. By setting the relative deadline to be $D_i = W_i/m = L_i$, every semi-clairvoyant scheduler will require a speed augmentation of $2 - 1/m$ to have bounded competitiveness.

With the previous theorem in place, we cannot hope for a $(1+\epsilon)$-speed $O(1)$-competitive algorithm. To circumvent this hurdle, one could hope to show $O(1)$-competitiveness by either using more resource augmentation or by making an assumption on the input. Intuitively, the hardness comes from having a relative deadline D_i close to $\max\{L_i, W_i/m\}$. In practice, this is unlikely to be the case. We show that so long as $D_i \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$ then there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm.

Theorem 1.2. *If $(1+\epsilon)(\frac{W_i - L_i}{m} + L_i) \leq D_i$ for every job J_i, there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

We note that this immediately implies the following corollaries. One with no assumptions on the input and one for "reasonable jobs."

Corollary 1.3. *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

Corollary 1.4. *There is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive for maximizing throughput if $(W_i - L_i)/m + L_i \leq D_i$ for all jobs J_i.*

The assumption on the job deadline is reasonable as there exists inputs for which even the optimal semi-non-clairvoyant scheduler has unbounded performance if the deadline is any smaller.

For the general profit scheduling problem, we can make the following assumption. For all jobs J_i its general profit function satisfies $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some $x_i^* \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$. This assumption states that there is no additional benefit for completing a job J_i before time x_i^*; this is the natural generalization of the assumption in the throughput case. Using this, we show the following.

Theorem 1.5. *If for every job J_i it is the case that $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some value of $x_i^* \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$, there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for the general profit objective.*

Corollary 1.6. *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing general profit.*

2 ALGORITHM FOR JOBS WITH DEADLINES

Here we present an algorithm S for jobs with deadlines and profits for which Theorem 1.2 holds.

On every time step, the algorithm S must decide which jobs to schedule and which ready nodes of each job to schedule. When a job J_i arrives, S calculates a value, n_i — the number of processors "allocated" to J_i. On any time step when S decides to run J_i, it will always allocate n_i processors to J_i. In addition, since S is semi-non-clairvoyant, it is unable to distinguish between ready nodes of J_i; when it decides to allocate n_i nodes to J_i, it arbitrarily picks n_i ready nodes to execute if more than n_i nodes are ready.

In the assumption of Theorem 1.2, each job follows the condition that $(1+\epsilon)(\frac{W_i - L_i}{m} + L_i) \leq D_i$ for some positive constant ϵ.

We define the following **constants**. Let $\delta < \epsilon/2$, $c \geq 1 + \frac{1}{\delta\epsilon}$ and $b = (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$ be fixed constants. For each job J_i, the algorithm S calculates $n_i = \frac{(W_i - L_i)}{\frac{D_i}{1+2\delta} - L_i}$, where n_i is the number of processors S will give to J_i if it decides to execute J_i on a time step.

Let $x_i = \frac{W_i - L_i}{n_i} + L_i$, which is the number of time steps to complete J_i on n_i dedicated processors (regardless of the order the nodes are executed in). Therefore, job J_i can meet its deadline if it is given n_i dedicated processors for x_i time steps during $[r_i, d_i]$.

We define a ***processor step*** as a unit of time on a single processor and the ***density*** of a job as $v_i = \frac{p_i}{x_i n_i}$. Note that this is a non-standard definition of density. We define the density as $\frac{p_i}{x_i n_i}$ instead of $\frac{p_i}{W_i}$, because we think of J_i requiring $x_i n_i$ processor steps to complete by S. Thus, this definition of density indicates the potential profit per processor step that S can obtain by executing job J_i.

The scheduler S maintains jobs that have arrived but are unfinished in two priority queues. Queue Q stores all the jobs that have been *started* by S. Queue P stores all the jobs that have arrived but have not been started. In both queues, the jobs are sorted according to the density from high to low.

Job Execution: At each time step t, S picks a set of jobs in Q to execute in order from highest to lowest density. If a job J_i has been completed or if its absolute deadline d_i has passed ($d_i > t$), S removes the job from Q. When considering job J_i, if the number of unallocated processors is at least n_i the scheduler assigns n_i processors to J_i for execution. Otherwise, it continues on to the next job. S stops this procedure when either all jobs have been considered or when there are no remaining processors to allocate.

A job J_i is δ-**good** if $D_i \geq (1+2\delta)x_i$. A job is δ-**fresh** at time t if $d_i - t \geq (1+\delta)x_i$. For any set T of jobs, let the set $A(T, v_1, v_2)$ contains all jobs in T with density within the range $[v_1, v_2)$. We define $N(T, v_1, v_2) = \sum_{J_i \in A(T, v_1, v_2)} n_i$. This is the total number of

processors that S allocates to jobs in $A(T, v_1, v_2)$. We say that the set of job $A(T, v_1, v_2)$ *requires* $N(T, v_1, v_2)$ processors.

Adding Jobs There are two types of events that may cause S to add a job to Q. Either a job arrives or S completes a job. When a job J_i arrives, S adds it to queue Q if it satisfies the following:

(1) J_i is δ-good;
(2) For all job $J_j \in Q \cup J_i$ it is the case that $N\left(Q \cup J_i, v_j, cv_j\right) \leq bm$. In words, the total number of processors required by jobs in $Q \cup J_i$ with density in the range $[v_j, cv_j)$ is no more than bm.

If these conditions are met, then J_i is inserted into queue Q; otherwise, job J_i is inserted into queue P. When a job is added to Q, we say that the job is ***started*** by S.

When a job completes, S considers the jobs in P from highest to lowest density but first removes all jobs with absolute deadlines that have passed. Then S checks if a job J_i in P can be moved to queue Q by checking if job J_i is δ-fresh and meets condition (2) from above. If both are true, then J_i is moved from queue P to Q.

Remark: Note that the Scheduler S pre-computes a fixed number of processors n_i assigned to each job; this may seem strange. We chose this design because n_i is approximately the minimum number of dedicated cores job J_i requires to complete by $\frac{D_i}{1+2\delta} \rightarrow D_i$, without knowing J_i's the DAG structure.

3 ALGORITHM FOR GENERAL PROFIT

The algorithm S' for jobs with general profit functions is similar to S. Due to space, we briefly sketch it and point out the differences.

Assigning cores, deadlines and slots to jobs: When a job J_i arrives, S' calculates a relative deadline D_i and a set of time steps I_i with n_i processors. I_i are the only time steps in which J_i is allowed to run. In each time step t in I_i, we say that J_i is assigned to t.

Note that for the general profit problem, a job J_i has no deadline. Thus, S' computes a D_i by searching all the potential deadlines D to find the minimum valid deadline using a complicated process which we omit due to space. The set of time steps I_i is then determined using the chosen deadline D_i.

From the assumption in Theorem 1.5, for each job J_i the profit function stays the same until $x_i^* \geq (\frac{W_i - L_i}{m} + L_i)(1+\epsilon)$. We set $n_i = \frac{W_i - L_i}{x_i^*/(1+2\delta) - L_i}$, where $\delta < \epsilon/2$. We define its the ***density*** as $v_i = \frac{p_i(D_i)}{x_i n_i} = \frac{p_i(D_i)}{W_i + (n_i - 1)L_i}$, where $x_i := \frac{W_i - L_i}{n_i} + L_i$.

Executing jobs: This procedure is similar to S, with the only difference that S' only picks jobs to execute that have been assigned to time step t.

ACKNOWLEDGMENTS

This work is supported in part by a Google Research Award, a Yahoo Research Award and NSF grants CCF-1617724, CCF-1150036 and CCF-1340571.

REFERENCES

[1] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. 2006. Adaptive Task Scheduling with Parallelism Feedback. In *PPoPP*.
[2] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. 2007. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*.
[3] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallel DAG Jobs Online to Minimize Average Flow Time. In *SODA '16*. 176–189.
[4] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. 2011. Competitive Algorithms for Due Date Scheduling. *Algorithmica* 59, 4 (2011), 569–582.
[5] Sanjoy Baruah. 2014. Improved Multiprocessor Global Schedulability Analysis of Sporadic DAG Task Systems. In *ECRTS 2014*. 97–105.
[6] Sanjoy Baruah. 2015. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*. 1323–1328.
[7] Sanjoy Baruah. 2015. Federated Scheduling of Sporadic DAG Task Systems. In *IPDPS 2015*. 179–186.
[8] Sanjoy Baruah. 2015. The federated scheduling of systems of conditional sporadic DAG tasks. In *EMSOFT 2015*. 1–10.
[9] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. 2015. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *ECRTS 2015*. 222–231.
[10] Sanjoy K. Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, Dennis Shasha, and Fuxing Wang. 1992. On the Competitiveness of On-Line Real-Time Task Scheduling. *Real-Time Systems* 4, 2 (1992), 125–144.
[11] Sanjoy K. Baruah, Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, and Dennis Shasha. 1991. On-line Scheduling in the Presence of Overload. In *Symposium on Foundations of Computer Science*. 100–110.
[12] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. 2013. Feasibility analysis in the sporadic dag task model. In *ECRTS*.
[13] Colin Campbell and Ade Miller. 2011. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press.
[14] Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. 2007. Provably Efficient Online Non-clairvoyant Adaptive Scheduling. In *IPDPS*. http://research.microsoft.com/apps/pubs/default.aspx?id=176942
[15] Sungjin Im and Benjamin Moseley. 2016. General Profit Scheduling and the Power of Migration on Heterogeneous Machines. In *SPAA '16*.
[16] Intel. 2013. Intel CilkPlus. (Sep 2013). https://www.cilkplus.org/.
[17] Bala Kalyanasundaram and Kirk Pruhs. 2000. Fault-Tolerant Real-Time Scheduling. *Algorithmica* 28, 1 (2000), 125–144.
[18] Bala Kalyanasundaram and Kirk Pruhs. 2000. Speed is as powerful as clairvoyance. *J. ACM* 47, 4 (2000), 617–643.
[19] Gilad Koren and Dennis Shasha. 1994. MOCA: A Multiprocessor On-Line Competitive Algorithm for Real-Time System Scheduling. *Theor. Comput. Sci.* 128, 1&2 (1994), 75–97.
[20] Gilad Koren and Dennis Shasha. 1995. Dover: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems. *SIAM J. Comput.* 24, 2 (1995), 318–339.
[21] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Analysis of Global EDF for Parallel Tasks. In *ECRTS '13*.
[22] Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, and Abusayeed Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *ECRTS '14*. 85–96.
[23] Brendan Lucier, Ishai Menache, Joseph Naor, and Jonathan Yaniv. 2013. Efficient online scheduling for deadline-sensitive jobs: extended abstract. In *SPAA '13*. 305–314.
[24] Lin Ma, R.D. Chamberlain, and K. Agrawal. 2014. Performance modeling for highly-threaded many-core GPUs. In *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*. 84–91.
[25] OpenMP. 2013. OpenMP Application Program Interface v4.0. (July 2013). http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
[26] Kirk Pruhs and Clifford Stein. 2010. How to Schedule When You Have to Buy Your Energy. In *APPROX 2010, and RANDOM 2010*. 352–365.
[27] James Reinders. 2010. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media.
[28] Julien Robert and Nicolas Schabanel. 2008. Non-clairvoyant scheduling with precedence constraints. In *SODA (SODA '08)*. 491–500.
[29] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. 2014. Parallel Real-Time Scheduling of DAGs. *IEEE Trans. Parallel Distrib. Syst.* 25, 12 (2014), 3242–3252.
[30] Daniel D Sleator and Robert E Tarjan. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM* 28, 2 (1985), 202–208.
[31] Ola Svensson. 2010. Conditional hardness of precedence constrained scheduling on identical machines. In *STOC 2010*. 745–754.
[32] Gerhard J. Woeginger. 1994. On-Line Scheduling of Jobs with Fixed Start and End Times. *Theor. Comput. Sci.* 130, 1 (1994), 5–16.