Approximate throughput maximization in scheduling of parallel jobs on hypercubes*

Ondřej Zajíček

Institute of Mathematics, AS CR
Žitná 25, CZ-11567 Praha 1, Czech Republic
ondrej.zajicek@mff.cuni.cz

Abstract. *We study scheduling of unit-time parallel jobs on hypercubes. A parallel job has to be scheduled between its release time and deadline on a subcube of processors. The objective is to maximize the number of early jobs. We provide an efficient 1.5-approximation algorithm for the problem.*

1 Introduction

We study the scheduling of unit-time parallel jobs on a parallel machine with a hypercube topology of a processor network. Each job is specified by an integral release time and deadline, and the number of processors it needs, which is required to be a power of two, to respect the hypercube topology. The jobs have to be scheduled between their release times and deadlines and the goal is to maximize the number of jobs completed before their deadline.

If we consider scheduling of sequential jobs (i.e., jobs requiring a single processor) instead of parallel jobs, the problem is trivial. The natural algorithm always schedules the jobs with the smallest deadlines (among the available jobs). A standard exchange argument shows that this is an optimal schedule. Once parallel jobs are introduced, this no longer works.

Usually, parallel scheduling problems are NP-hard because they include some partitioning problem. Either partitioning the processors among the jobs, or partitioning the jobs into groups with the same total processing time is involved. In our case, the hypercube topology, where processors are connected to form a hypercube and jobs (having a size that is a power of two) are scheduled on appropriate subhypercubes, together with the restriction to unit processing times make these packing problems easy. If we were able to compute which jobs should be scheduled in each timeslot, we could always assign the chosen jobs to subcubes in a greedy manner from the largest job to the smallest one.

* This research was partially supported by Institute for Theoretical Computer Science, Prague (proj. 1M0545 of MŠMT ČR), grant IAA100190902 of GA AV ČR and by Institutional Research Plan No. AV0Z10190503.

However, there is no known polynomial algorithm even to decide whether it is possible to schedule all jobs within their constraints (feasibility testing). Ye and Zhang [3] showed that it is possible to maximize the number of completed jobs if all the release times are equal. This was generalized to the case of nested intervals given by the release times and the deadlines, see [4].

For general release times and deadlines, there are positive results for the 'tall/small' model, in which only jobs that request one or all processors are allowed. Baptiste and Schieber [1] showed that feasibility testing in the 'tall/small' model is polynomially solvable. The article contains two algorithms for the 'tall/small' problem, see also [2] for an alternative proof. However, the maximization of the number of completed jobs is open even for two processors, which is a special case of the tall/small variant.

Our previous result [5] used the same model (scheduling of hypercubes with general release times and deadlines) but instead of an offline solution it presented an 1.6-competitive online algorithm. We refined some ideas from this result and extended it to take advantage of the offline setting to get an 1.5-approximation algorithm.

2 Preliminaries

The problem has a parameter m giving the number of machines. An instance of the problem consists of a set of n jobs. Each job J has an integral release time r_J, an integral deadline d_J and a size s_J (the number of requested processors). The numbers m and s_J are powers of two. As all times are integers and jobs' processing times are equal to one, instead of time we can consider timeslots (aligned unit-time intervals) and every job requests one timeslot.

We say that job J is feasible at timeslot T if $r_J \leq T$ and $T < d_J$. We say that job J is available at timeslot T if it is feasible and not scheduled yet. We say that job J is urgent at timeslot T if $d_J = T+1$. A schedule assigns to each processed job J find a timeslot T such that J is feasible at T, and s_J processors, so that no processor is assigned to two jobs at the same time.

The objective is to find a schedule maximizing the number of processed jobs.

We fix an ordering $\prec$ on jobs that is a strict linear ordering based on the ordering of deadlines, in a case of equal deadlines it is defined arbitrarily. For example, we take an ordering defined by formula $J_i \prec J_j \Leftrightarrow d_i < d_j \vee (d_i = d_j \wedge i < j)$. We suppose, w.l.o.g., that any algorithm chooses the $\prec$-minimal job from the available jobs of the same size when it needs to choose one job of that size.

We use ALG to denote the analyzed algorithm and OPT to denote an optimal offline algorithm. Jobs of size m are called *max-jobs*, smaller jobs are called *non-max jobs*. Jobs of size 2^i are called *i-jobs* (where i is some integer).

3 Algorithm

The algorithm is based on the online algorithm from [5], which is a memoryless online algorithm that in each timeslot examines a set of available jobs and chooses the maximal subset of jobs to process in that timeslot according to these rules (in the order of importance):

- Prefer more smaller jobs over one bigger job.
- Prefer an urgent job over an non-urgent job.
- Prefer a bigger job over a smaller job.
- Prefer $\prec$-minimal jobs between jobs of the same size.

We call such subset of available jobs a *T-preferred set*.

The algorithm is modified so that in the timeslots where a set of available jobs contains some max-jobs and exactly one non-max job M that has deadline smaller than all these available max-jobs but it is still non-urgent, the algorithm not only chooses to process the max-job with the smallest deadline (by rules 3 and 4) but also marks job M. If job M is scheduled later, the algorithm just clears the mark. But if job M expires later without being scheduled (when the algorithm processed the last timeslot before the deadline of the marked job), the algorithm examines all max-jobs scheduled from the time when job M was marked, chooses the one with the largest deadline and replaces it with job M (and also clears its mark). In that case we call job M *a replacer job*. Because the replaced max-job has a bigger deadline than replacer job M, it will reappear in a set of available jobs. The algorithm continues with processing the next timeslot (the timeslot that is equal to the deadline of job M). Note that when one job is marked, another job cannot be marked until the mark is cleared.

The algorithm is described by the following pseudocode representing a loop body. Global variables are T for current timeslot, A for a set of available jobs and MJ, MT for a marked job and its timeslot, other variables are local to the loop iteration.

1. Add jobs with release time T to set A.
2. Compute T-preferred set S from the set of available jobs A (specified below).
3. If set A contains exactly one non-max job M, which is not a member of set S, let $MJ := M$, $MT := T$ (mark job M and timeslot T).
4. Remove jobs that are members of set S from set A, store set S as a schedule for timeslot T.
5. Remove jobs with deadline of $T + 1$ from set A.
6. If job MJ was removed in the previous step, examine the computed schedules from timeslot MT to timeslot T to find a scheduled max-job J with the largest deadline, change schedule for timeslot containing job J to contain job MJ instead, and add job J to the set A.
7. Repeat with $T := T+1$, until all jobs are processed and set A is empty.

To complete the description of the algorithm, it remains to describe how to compute a T-preferred set S. In timeslot T it is possible to schedule any set of jobs satisfying that each its member is available during T and a sum of sizes of its members is less than or equal to m. Let such a set be called *a T-schedulable set*.

Let us consider a set of all T-schedulable sets. First, we restrict ourselves to the T-schedulable sets that maximize the number of jobs. Second, we restrict ourselves to the sets that maximize the number of urgent jobs. And finally, we restrict ourselves to the sets that maximize the sum of the sizes of the jobs. Let the remaining schedulable sets be called *T-conforming sets*.

Lemma 1. *All T-conforming sets have the same number of jobs of specific sizes.*

Proof. Let us have two T-conforming sets S_1 and S_2 that have different number of i-jobs (w.l.o.g. S_1 contains more i-jobs than S_2) and the same number of smaller jobs. As both S_1 and S_2 have the same number of jobs and the same sum of sizes of jobs, the difference between number of i-jobs has to be an even number (otherwise it would not be possible to balance the sum of sizes by bigger jobs) and there has to be some j-job ($j > i$) in S_2 and not in S_1 (for the same reason). We can remove one j-job from S_2 and add two more i-jobs (that are in S_1 and not in S_2) and we still get a T-schedulable set, but with more jobs than S_1 (and S_2). As S_1 maximizes the number of jobs (between all T-schedulable sets), this is a contradiction. □

Let n_i be the number of i-jobs in any T-conforming set (this is well-defined by Lemma 1). We choose the T-preferred set as a set containing (for each i) n_i

$\prec$-smallest i-jobs from all i-jobs available during T. Obviously, the T-preferred set is also a T-conforming set.

Our algorithm needs to compute a T-preferred set for timeslot T. The T-preferred set can be constructed efficiently by the following procedure:

1. Sort available jobs (set A, input to the procedure) according to their job sizes in increasing order. In the case of a tie, $\prec$-smaller jobs are preferred.
2. Choose as many jobs as possible (the sum of the sizes of the chosen jobs is not allowed to exceed m) in the sorted order. Let C be the set of chosen jobs.
3. If all jobs were chosen, finish and return C. Otherwise, let X be the first job that was not chosen.
4. Find the smallest non-urgent job Y that is sufficiently large so that its removal from C makes enough space to be able to add X to C. In case of a tie, a $\prec$-bigger job is preferred.
5. If Y was not found in the previous step and X is urgent, then repeat the search but look for an urgent job instead of a non-urgent job.
6. If Y was found in step 5 or 6, let $C' = C \setminus \{X\} \cup \{Y\}$, otherwise let $C' = C$.
7. Return C'.

Lemma 2. *Set C from the procedure can be transformed to any schedulable set that maximizes the number of jobs by replacing some jobs with jobs of the same size and at most one job of an arbitrary size with a job of size s_X.*

Proof. C is obviously a schedulable set that maximizes the number of jobs; therefore, it contains the same number of jobs as any schedulable set that maximizes the number of jobs; therefore, to reach such sets we may restrict to one-for-one job replacements. We may ignore replacements with jobs smaller than X because all such jobs are already in C. Replacements with bigger jobs are limited by the number of free machines. It is not possible to replace a job with a job larger than job X, otherwise there would be enough free machines to choose X in step 2. It is also not possible to replace two (smaller) jobs with jobs of the same size as job X, by the same argument. □

Theorem 1. *For time T, the procedure finds the T-preferred set.*

Proof. By Lemma 2, we can transform set C to the T-preferred set by some job replacements. There is no need for replacements between jobs of the same size because if there are k i-jobs in C', then they are k $\prec$-smallest available i-jobs. The remaining replacement (Y with X) is chosen to maximize the number of urgent jobs (X is the $\prec$-smallest between possible choices, if X is not an urgent job, then Y is neither) and remove smallest jobs to maximize the sum of sizes of jobs. □

4 Approximation ratio

We will use a charging scheme to prove the upper bound for the approximation ratio of ALG. A charging scheme is a set of rules for a specification of weighted edges between the set of jobs in ALG schedule and the set of jobs in OPT schedule to create a bipartite graph. This graph obeys some constraints: For each job in OPT schedule the sum of the weights of incident edges is exactly 1 and for each job in ALG schedule the sum of weights of incident edges is at most 1.5. These constraints (and the fact that this scheme specifies such a matching for OPT and ALG schedules of every instance) imply that the approximation ratio of the algorithm is at most 1.5.

We introduce some terminology. When there is an edge between two jobs with weight x we write that the job in OPT schedule *sends* x and the job in ALG schedule *receives* x. The charging scheme uses mainly two kinds of edges: *diagonal edges* and *vertical edges*. A diagonal edge is an edge from a job in OPT schedule to the same job in ALG schedule in a different timeslot. A vertical edge is an edge from a job in OPT schedule to any job in ALG schedule in the same timeslot.

We use *a job* in two slightly different meanings. First, there is a particular job from an instance of a problem. Second, the job is scheduled by a particular schedule to some machines and some timeslot. The position occupied by some job in the particular schedule is also called the job. Specifically, we use *ALG-job* for the position of a job in ALG schedule and *OPT-job* for the position of a job in OPT schedule. Obviously, the charging edges do not connect jobs in the first sense, but ALG-jobs and OPT-jobs.

A job not scheduled by ALG (but possibly scheduled by OPT) is called *an unscheduled job*. A job scheduled by OPT and either not scheduled by ALG during that or earlier timeslots (but possibly scheduled later) or scheduled by replacement (a replacer job) is called *a free job*. The motivation for such definition is that a free job is a job that is available for ALG at the timeslot in which it is scheduled by OPT. Because a decision to do a replacement is done later in the run of the algorithm, replacer jobs (scheduled by OPT) are also counted as free jobs. An important property of free job J is that the relevant timeslots (from release time of job J to the timeslot when job J is scheduled by OPT) in ALG schedule do not contain enough free machines to schedule job J.

If there is a max-job in ALG schedule and in the same timeslot there is only one non-max free job (regardless of the number of non-free jobs) in OPT schedule, then we call this non-max free job *a red job*. Note that in such case the red job is the only non-max job available to ALG (otherwise ALG would also choose it by rule 1) and therefore it is marked by ALG and is a candidate to be a replacer job. Other free jobs are called *white jobs*, non-free jobs (scheduled first by ALG and later by OPT, or scheduled in the same timeslot by ALG and OPT) are called *black jobs*.

The charging scheme is specified as follows: Each black job charges one diagonal edge (to the same job in ALG schedule), each white job charges one vertical edge (upwards to an unspecified job in the same timeslot). Each red job charges $1/2$ diagonally and $1/2$ vertically. We will specify exact rules for a distribution of vertical edges to ALG-jobs later.

Matching of i-jobs at timeslot T is a process that finds a maximal matching between a set of i-jobs in ALG schedule of timeslot T and a set of white i-jobs in OPT schedule of timeslot T. If there is a job scheduled at timeslot T by both ALG and OPT, then it is matched with itself, remaining jobs are matched arbitrarily with one restriction: any red jobs J in ALG schedule are matched at the end, only when no other jobs remain. Some i-jobs may be left unmatched in ALG or OPT schedule, but not in both schedules.

We will use modified variants of lemmas from [5]:

Lemma 3. *If non-red ALG-job A (scheduled at some timeslot T) is matched with OPT-job B, then A receives nothing diagonally (from OPT-job A).*

Proof. If job A is white, it is obvious (white jobs does not charge diagonally). If job A is black, we prove it by contradiction. Suppose ALG-job A receives diagonally from (black) OPT-job A. Jobs A and B have to be different jobs, because OPT-job B is white. Because B is a white job, it follows that ALG did not schedule B before or at timeslot T. Because A is black, OPT scheduled A after timeslot T. Thus both A and B were available to both ALG and OPT at timeslot T, but ALG scheduled A and didn't schedule B and OPT scheduled B and didn't schedule A. This is a contradiction because A and B are jobs of the same size and both algorithms choose the $\prec$-minimal jobs from available jobs of the same size. □

Lemma 4. *For every timeslot it is possible to find a distribution of weight of all incoming vertical edges between ALG-jobs of the timeslot such that every job in ALG schedule can be categorized to at least one of these classes:*

- *Class C (common): The job receives at most* $1/2$ *vertically.*
- *Class M (matched): The job receives* 1 *vertically from the matched job. If the job is non-red, it could also receive* $1/2$ *vertically from another job.*
- *Class U (urgent): The job is urgent and receives at most* 1 *vertically.*
- *Class R (replacer): The job is a replacer job and receives at most* 1 *vertically.*

Proof. The proof is done independently for each timeslot. We show that for each free job in OPT schedule we find the same job or two other jobs in ALG schedule (in the same timeslot). Let T be any fixed timeslot. We use ALG_T (and OPT_T) schedule for ALG (and OPT) schedule restricted to timeslot T.

If there is no job in ALG_T schedule, then all jobs in OPT_T schedule have to be black, because any free OPT_T job could also be scheduled by ALG at T. So suppose there are some jobs in ALG_T schedule and the biggest job among them is an i-job. Jobs smaller than 2^i will be called small jobs. It is easy to see that there is no more than one small free job in OPT_T schedule—otherwise ALG should schedule two (or more) small jobs instead of the i-job. We distinguish two cases: one small free job and no small free job.

Case 1: There is exactly one small free job J in OPT_T schedule. First we match i-jobs in T. We split the timeslot in ALG_T schedule to slots of size 2^i. In each slot there is either one i-job or more small jobs (there is neither an empty slot nor a slot with one small job, otherwise the free space in that slot is large enough that ALG should schedule the job J in it). Now we assign those slots to OPT_T free jobs. The idea is that each OPT_T free i-job gets one slot and larger free jobs get proportionally more slots. Slots with matched i-jobs are assigned to matched OPT_T free i-jobs. If there are remaining OPT_T free i-jobs, they get slots with more small jobs. If we disregard job J then the rest is correct: matched ALG_T i-jobs are class M jobs, smaller jobs (assigned together to one job) are class C as well as remaining i-jobs assigned together to larger jobs. Unused ALG_T jobs may be class C as they receive nothing vertically. Now we find the assignment for job J. There are two cases:

Case 1.1: There is at least one slot with more small jobs. Then we assign it in the first place to job J (and those small jobs are class C) and the lemma holds.

Case 1.2: There are only i-jobs in ALG_T schedule (and one i-job called job K is assigned to job J). We have three cases distinguished by the structure of OPT_T schedule.

Case 1.2.1: There is at least one free i-job (job L) in OPT_T schedule. Then job L is matched with some ALG_T i-job (job L'). Job L' receives 1 vertically from job L; hence, it is a class M job and it is non-red

(because there is at most one red job in ALG_T schedule and there are more i-jobs in ALG_T schedule than in OPT_T schedule, therefore the eventual red job left unmatched as it would be matched at the end), therefore it can receive additional $1/2$ from job J. Job K receives remaining $1/2$ from job J, is a class C job and the lemma holds.

Case 1.2.2: There is no free i-job in OPT_T schedule but there are some larger free jobs. Then there are two unused slots in ALG_T schedule, because the sum of sizes of larger OPT_T jobs is a multiple of 2^{i+1} and the number of ALG_T i-jobs assigned to them is even. Therefore, there are at least two ALG_T i-jobs available, they receive $1/2$ from job J and are class C.

Case 1.2.3: Job J is the only free job in OPT_T schedule. If there are more than one ALG_T i-job then two of them receive $1/2$ and are class C. If there is only one job M, then M has to be max-job, because there is no empty slot (ALG_T is full of i-jobs). In that case job J (which is not a max-job because it is a small job) is a red job and therefore charges just $1/2$ to job M and job M is class C.

Case 2: There is no small free job in OPT_T schedule. Let j-jobs be the smallest free OPT_T jobs, obviously $j \geq i$. First we match j-jobs (which does nothing if $j > i$). We split timeslot T in ALG_T schedule to slots of size 2^j. No such slot is empty (otherwise, ALG should schedule some free j-jobs scheduled by OPT at T). At most one slot is not full (because job sizes are powers of two we can always pack jobs from two half-empty slots to make one slot empty or full). Now we assign the slots to OPT_T free jobs as we did in the first case. If we have only slots with either one j-job or with more smaller jobs then it is the same argument as in first case (even easier because there is no job J). But the one non-full slot can contain only one job (job N), which is smaller than j-job. In that case job N has to be urgent or a replacer job; otherwise, ALG should schedule some free j-job instead of job N, by rule 3. Therefore, job N is a class U or class R job and the slot with job N may be used much like a slot with two jobs. Even in this case the lemma holds. □

Theorem 2. *The approximation ratio of ALG is at most* 1.5.

Proof. We described the charging scheme earlier. To complete the proof it remains to show that each ALG job receives at most 1.5 of the charged edges. According to Lemma 4, it is possible to distribute vertical edges between ALG jobs in such a way that ALG jobs can be divided to four classes C, M, U, and R. Class C jobs receive at most $1/2$ vertically (by definition) and at most 1 diagonally (as every job). Non-red class M jobs receive at most 1.5 vertically (by definition) and nothing diagonally (by Lemma 3), Red class M jobs receive at most 1 vertically (by definition) and at most $1/2$ diagonally. Class U and class R jobs receive at most 1 vertically (by definition) and at most $1/2$ diagonally if they are red. They receive nothing diagonally from a black job because they are never black jobs (class R jobs by definition and class U jobs because they are urgent and therefore they cannot be scheduled later by OPT). Therefore, each ALG job receives at most 1.5. □

5 Conclusion

We addressed the offline scheduling problem of unit-time parallel jobs on hypercubes to maximize the number of early jobs. We have presented an efficient 1.5-approximation algorithm for the problem. The result extends our previously published result [5].

A natural question is whether it is possible to find the optimal solution to the problem (or at least some restricted variant of the problem, like the tall/small scheduling) in polynomial time. Another interesting question is what approach should be used for the weighted variant of the problem, where every job has a weight and the objective is to maximize the sum of weights of early jobs.

References

1. P. Baptiste and B. Schieber: *A note on scheduling tall/small multiprocessor tasks with unit processing time to minimize maximum tardiness.* J. Sched., **6**, 2003, 395–404.
2. C. Dürr and M. Hurand: *Finding total unimodularity in optimization problems solved by linear programs.* In: Proc. 13th European Symp. on Algorithms (ESA), volume 4168 of Lecture Notes in Comput. Sci., Springer, 2006, 53–64.
3. D. Ye and G. Zhang: *Maximizing the throughput of parallel jobs on hypercubes.* Inform. Process. Lett., **102**, 2007, 259–263.
4. O. Zajíček: *A note on scheduling parallel unit jobs on hypercubes.* Int. J. on Found. Comput. Sci., **20** (2), 2009, 341–349.
5. O. Zajíček, J. Sgall, and T. Ebenlendr: *Online scheduling of parallel jobs on hypercubes: Maximizing the throughput.* Technical report ITI Series 2009-481, Charles University, Prague, 2009; to appear in PPAM 2009 Proceedings.