Online Scheduling of Parallel Jobs on Hypercubes: Maximizing the Throughput

Ondřej Zajíček[1], Jiří Sgall[2], and Tomáš Ebenlendr[1]

[1] Institute of Mathematics, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic.
[2] Dept. of Applied Mathematics, Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, CZ-11800 Praha 1, Czech Republic.

Abstract. We study the online problem of scheduling 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 a 1.6-competitive algorithm for the problem and prove that no deterministic algorithm is better than 1.4-competitive.

1 Introduction

We consider scheduling of parallel jobs on hypercubes with the objective to maximize the number of jobs completed before their deadline. We focus on the case where all processing times are equal to 1. In this case, 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.

In the online setting, the jobs arrive over time: Each job arrives at its release time; at this time its complete specification is released. At each time step we need to choose a subset of available jobs that are scheduled. Available jobs are those that are already released, not yet scheduled, and with a deadline strictly larger than the current time. The total number of processors required by the chosen jobs needs to be at most the size of the hypercube.

The hypercube topology restricts the actual assignment of parallel jobs: The processors are organized as a hypercube and each job has to be scheduled on a subcube of the hypercube. However, since we consider only jobs with unit processing times, this restriction is equivalent to the constraint that job sizes as well as the total number of processors are powers of two. Once the total processor requirement is at most the number of processors, we can always assign the chosen jobs to subcubes in a greedy manner from the largest job to the smallest one.

Our results. We present a 1.6-competitive algorithm for this problem. In two special cases we show that the algorithm is 1.5-competitive. The first special case excludes jobs that require the whole hypercube. The second special case is that of tall/small jobs, where each job may require either the whole hypercube or a single processor. We show that the analysis of this algorithm is tight. Our algorithm is memoryless, i.e., its action at each time depends only on the currently available jobs. We prove that no deterministic algorithm is better than 1.4-competitive.

This is true even on a machine with two processors, which is a subcase of the tall/small special case.

Related results. If we restrict ourselves to sequential jobs (i.e., jobs requiring a single processor), 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. We need to find a rule to choose, for example, between urgent parallel jobs and sequential jobs with large deadlines.

A simple approach to similar problems is the greedy algorithm. This works even if we allow both parallel jobs and weights. In each time step, we schedule a set of jobs with maximal total weight from the available jobs. A standard charging argument shows that this algorithm is 2-competitive. For each job in the optimal schedule, charge its weight to the timeslot in the online schedule where it is scheduled; if it is not scheduled, charge it to the same timeslot. To each timeslot in the online schedule, we charge at most twice the total weight of the jobs scheduled by the online algorithm: First, we may charge each job to itself. Second, we charge the jobs scheduled by the optimum at the same time, but not scheduled by the online algorithm; these jobs are available, thus their total weight is at most the weight of the jobs scheduled greedily. Summing over all the timeslots, 2-competitiveness follows. Improving the competitive ratio below 2 for this general problem is a challenging open problem.

The complexity of the offline problem is not known. Typically, 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. The hypercube constraint and the restriction to unit processing times make these packing problems trivial. Nevertheless, polynomial algorithms are known only for a couple of special cases. One can maximize the number of completed jobs if all the release times are equal, see [3]. 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, the only positive result exists for the tall/small case studied in [1], see also [2] for an alternative proof; however, this gives only an algorithm for testing if all jobs can be completed. The throughput maximization is open even for the case of two processors, which is a special case of the tall/small case.

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. In the online variant of the problem, at the timeslot T, we get a knowledge of all jobs J with $r_J = T$ and we have to decide which jobs start to process at timeslot T.

We consider a variant of the generalized problem where all jobs have unit processing time ($p_J = 1$) and their release times and deadlines are integers. We also restrict the size s_J of jobs and the number of processors m to be a power of two. As mentioned in the introduction, this is equivalent to the requirement that each job is scheduled on a subcube of the hypercube with m processors.

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 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 number).

2 Algorithm

We want an algorithm that chooses from possible schedules according to these four rules, in the order of importance, because such rules lead to invariants used in the proof of the competitive ratio:

1. Prefer more smaller jobs over one bigger job.
2. Prefer an urgent job over a non-urgent job.
3. Prefer a bigger job over a smaller job.
4. Prefer $\prec$-minimal jobs among the jobs of the same size.

It is easy to convert these rules to a memoryless algorithm that (for each timeslot) examines a set of currently available jobs and chooses its maximal schedulable subset (a set such that the sum of the sizes of its members is less than or equal to m) satisfying these rules (e.g., if there is a non-urgent job in the chosen subset, then there is no urgent job of the same or smaller size outside of the chosen subset). The chosen subset will be scheduled in that timeslot.

Lemma 1. *The competitive ratio of the algorithm is at least* 1.6*.*

Proof. We construct an instance on four machines. (For more machines, it can be easily scaled up.) One 1-job X with deadline 3 and one 2-job A with deadline 5 are released at time 1. The algorithm chooses job A at time 1 (by rule 3). Consequently, two 1-jobs Y and Z with deadline 3 and four 0-jobs B, C, D, E with deadline 4 are released at time 2 and the algorithm chooses four 0-jobs (by rule 1) and loses all three 1-jobs (X, Y, Z). OPT schedules all jobs: three 1-jobs in the first two timeslots, four 0-jobs in timeslot 3 and the remaining 2-job in timeslot 4. The proof is summed up in Figure 1. □

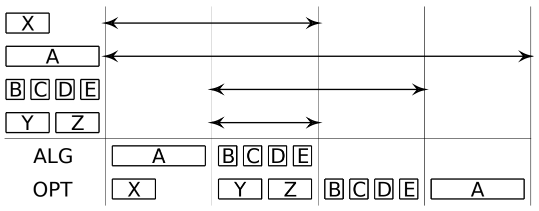

Fig. 1. The proof of the lower bound in a general case.

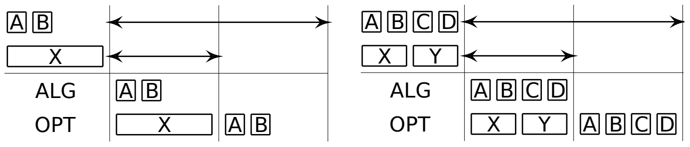

Fig. 2. The proof of the lower bound in the restricted cases.

Lemma 2. *The competitive ratio of the algorithm is at least* 1.5 *in the tall/small case and in the non-max case.*

Proof. We construct two instances on four machines. Some urgent larger jobs and more non-urgent smaller jobs are released at time 1. The algorithm chooses more non-urgent smaller jobs and loses the urgent larger jobs. Details of sizes and counts of jobs are summed up in Figure 2, the left-hand side is for the tall-small case and the right-hand side is for the non-max case. The proof can be easily scaled up for more machines. □

3 Competitive ratio

We prove the upper bound for the competitive ratio of ALG using a charging scheme. When we consider jobs in ALG and OPT schedules as two sets of vertices, then the charging scheme is a set of rules for a specification of weighted edges between these two sets 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 1.6 (or 1.5 in the restricted cases). These constraints (and the fact that this scheme specifies such a matching for OPT and ALG schedules of every instance) imply that the competitive ratio of the algorithm is at most 1.6 (or 1.5 in the restricted cases).

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. A job not scheduled by ALG (but possibly scheduled by OPT) is called *an unscheduled job*. A job scheduled by OPT and not scheduled by ALG during that or earlier timeslots (but possibly scheduled later) is called *a free job* because it is available for ALG at the timeslot in which it is scheduled by OPT.

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.

If there is a max-job in ALG schedule and in the same timeslot there is only one non-max free job in OPT schedule, then we call this non-max free job *a red job*. Other free jobs are called *white jobs*, non-free jobs (scheduled first by ALG and later by OPT) are called *black jobs*. In the first part of proof we define charging for white and black jobs, in the second part for red jobs.

The charging scheme is specified as follows: Each black job charges one diagonal edge (forwards to the same job in ALG schedule) and each white job charges one vertical edge (upwards to an unspecified job in the same timeslot). 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 arbitrary 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.

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

Proof. 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. $\square$

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.*
– *Class U (urgent): The job is urgent and receives at most* 1 *vertically.*
– *Class S (special): The job is scheduled at a timeslot that is full of jobs of the same size in ALG schedule. Furthermore, it is a non-max job and at most one job per timeslot can be the class S job. The job receives* 1 *vertically from the matched job and* $1/2$ *vertically from another job.*

Proof. The proof is done independently for each timeslot. We show that for each white 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 white 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 white 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 white job and no small white job.

Case 1: There is exactly one small white 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 white jobs. The idea is that each OPT_T white i-job gets one slot and larger white jobs get proportionally more slots. Slots with matched i-jobs are assigned to matched OPT_T white i-jobs. If there are remaining OPT_T white 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 white 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 can receive additional $1/2$ from job J and become a class S job. Additional constraints for a class S job also hold: L' is a non-max job, as otherwise ALG should have scheduled the two white jobs J and L by rule 1. There is only one class S job, because there is only one J job. 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 white i-job in OPT_T schedule but there are some larger white 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 white 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). This case cannot appear, as job J, which is not a max-job because it is a small job, would be a red job and not a white job.

Case 2: There is no small white job in OPT_T schedule. Let j-jobs be the smallest white 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 white 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 white 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: otherwise, ALG should schedule some white j-job instead of job N, by rule 3. Therefore, job N is a class U job and the slot with job N may be used much like a slot with two jobs. Even in this case the lemma holds. □

Lemma 5. *Let $I(J)$ (for any non-black job J) be a time interval starting by the timeslot when the time when job J was scheduled by OPT and ending by the last timeslot when J was available for ALG (it was scheduled by ALG or it was just before the deadline for unscheduled job J). Then $I(J)$ for all red jobs J do not overlap.*

Proof. Suppose two such intervals overlap. Let T be the first timeslot in their intersection. In timeslot T both jobs are available for ALG, ALG should schedule both jobs together (as they are non-max) but it scheduled one max-job instead. □

Lemma 6. *Let T be a timeslot when some unscheduled red job J is urgent. Then either there is an ALG job receiving at most 1 in timeslot T, or there are at least four ALG jobs in timeslot T.*

Proof. We split timeslot T to slots of the same size as job J. In each slot there is either one urgent job (or part of that job) or more than one job because if there is an empty slot or a slot with a non-urgent job then ALG should schedule job J in that slot instead. Because J is a non-max job there are at least two slots. We choose two slots such that at least one of them does not contain a class S job (or its part), which is possible, because any class S job is a non-max job.

Case 1: Both slots contain more jobs. Then there are at least four jobs and the lemma holds.

Case 2: One slot contains more jobs and the other slot contains an urgent job K. Then job K is not a class S job: otherwise all ALG jobs would have the same size and all the slots would have to be full. If job K is class C, P or U then it receives at most 1 vertically and nothing diagonally (because K is urgent) and the lemma holds.

Case 3: Both slots contains one urgent job. At least one of the slots does not contain class S job; hence, it contains a class C, P or U job, which receieves at most 1 and the lemma holds as in case 2. □

Lemma 7. *Let OPT job J be a scheduled red job. Then ALG job J is not a class S job.*

Proof. Suppose that ALG job J is scheduled at timeslot T and is a class S i-job. In that case timeslot T of ALG schedule is filled by i-jobs and there is one smaller job in OPT schedule at timeslot T, as these are the assumptions of case 1.2.1 in Lemma 4 that are needed to classify a job as a class S job. Therefore, for matching used for classification of jobs there are more i-jobs in ALG schedule than in OPT schedule and by the definition of matching in that case job J as only red job remains unmatched and thus cannot be classified as class S job. Contradiction. □

Theorem 1. *The competitive ratio of ALG is at most* 1.6 *in a general case and at most* 1.5 *in the tall/small case and in the non-max case.*

Proof. We described the charging scheme for black and white OPT jobs earlier. To complete the proof it remains to describe the charging scheme for red OPT jobs and to show that each ALG job receives at most 1.6 (or 1.5). 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, P, U and S. Class C jobs receive at most 1/2 vertically (by definition) and at most 1 diagonally (as every job). Class M jobs receive at most 1 vertically (by definition) and nothing diagonally (by Lemma 3). Class U jobs receive at most 1 vertically (by definition) and nothing diagonally because they are urgent and therefore they cannot be scheduled later by OPT. Class S jobs receive at most 1.5 vertically and nothing diagonally (by Lemma 3). Therefore, if there is no red job, then each ALG job receives at most 1.5.

Now we describe charging scheme for red jobs. For each red (OPT) job J, there are two cases whether job J is scheduled or not by ALG. There is also a max-job in ALG schedule (from the definition of red jobs) called job K. In the first case, job J is scheduled, and it sends 1/2 vertically to ALG job K, the only ALG job in that timeslot and 1/2 to itself in ALG schedule (ALG job J). In the second case, job J is unscheduled, and it sends 0.6 vertically to ALG job K and 0.4 to the timeslot T when job J is urgent. According to Lemma 6, either there is a ALG job in timeslot T receiving at most 1 from white and black jobs which then receives 0.4 from job J, or there are some four jobs scheduled at timeslot T and they receive 0.1.

We showed that without red jobs the theorem holds. Now we show that even if we add charging of red jobs, then the upper bounds hold. Because of Lemma 5, the charging from different red jobs does not mix and we can deal with each red

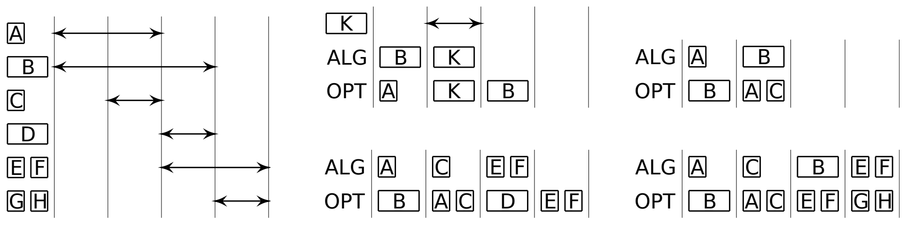

Fig. 3. The instance for the lower bound and the four cases in the proof.

job independently. In the first case of the definition of charging, job K received nothing vertically from white jobs (as there is no such job in the same timeslot) and at most 1 diagonally (as every job). Therefore we can add 1/2 from the red job. ALG job J is not class S job according to Lemma 7. Therefore, it receives at most 1 vertically from white jobs (because it is in class C, P or U) and nothing diagonally (because OPT job J is red and not black) and we can add 1/2 from the red job. In the second case, the argument for job K is the same as in the first case, but we add 0.6 and therefore it receives at most 1.6 and we can add 0.1 to each of some four jobs because without charging from red jobs they receive at most 1.5, with that they receive at most 1.6. Therefore, we shown that if there are no unscheduled red jobs, each ALG job receives at most 1.5, otherwise it receives at most 1.6 and that proves the first part of the theorem.

In the first restricted case (no max-jobs) it is obvious that there are no red jobs and therefore 1.5 is the upper bound for the competitive ratio. In the second restricted case (tall/small jobs) suppose that there is an unscheduled red i-job J, which is urgent at timeslot T. Because there are no smaller jobs, ALG schedule of timeslot T must be full of urgent i-jobs. As the algorithm does not specify any preferences for choosing urgent jobs of the same size, ALG might choose job J instead of one of these i-jobs (and similarly for other unscheduled red jobs). As both jobs are urgent, such choice would not affect the remaining schedule. In that case there would be no unscheduled red jobs and therefore upper bound 1.5 holds. But even if ALG did not choose job J then there is the same number of jobs scheduled by ALG and therefore upper bound 1.5 holds. That proves the second part of the theorem. □

4 Lower bound

Theorem 2. *The competitive ratio of every deterministic algorithm for online scheduling of parallel jobs on hypercubes is at least* 1.4*.*

Proof. We fix a deterministic algorithm. We consider an instance on two machines; it can be easily scaled up for more machines. The instance and the proof are summed up in Figure 3.

We start with two jobs A ($r_A = 1$, $d_A = 3$, $s_A = 1$) and B ($r_B = 1$, $d_B = 4$, $s_B = 2$). If the algorithm schedules job B in the first timeslot, then we extend the instance with an urgent job K ($r_K = 2$, $d_K = 3$, $s_K = 2$) and finish. The algorithm might schedule job A or job K, but loses the other job and it is 1.5-competitive (as OPT schedules all jobs). This is the first case in Figure 3.

Otherwise, the algorithm schedules job A in the first timeslot and OPT schedules job B. We extend the instance with an urgent job C ($r_C = 2$, $d_C = 3$, $s_C = 1$). If the algorithm schedules job B in the second timeslot, then it loses job A and it is 1.5-competitive (as OPT schedules all jobs). This is the second case in Figure 3.

Otherwise, the algorithm schedules job C in the second timeslot and OPT schedules jobs A and C. We extend the instance with an urgent job D ($r_D = 3$, $d_D = 4$, $s_D = 2$) and two smaller jobs E and F ($r_E = r_F = 3$, $d_E = d_F = 5$, $s_E = s_F = 1$), If the algorithm schedules jobs E and F for the third timeslot, then it loses jobs B and D and it is 1.5-competitive (as OPT schedules all jobs). This is the third case in Figure 3.

Otherwise, the algorithm schedules job B or D in the third timeslot and OPT schedules jobs E and F. We extend the instance with two urgent jobs G and H ($r_G = r_H = 4$, $d_G = d_H = 5$, $s_G = s_H = 1$), The algorithm schedules two jobs and loses the other two jobs from jobs E, F, G, H and it is 1.4-competitive, as OPT schedules all these four jobs during last two timeslots, but loses job D. This is the fourth case in Figure 3. □

Acknowledgements. This research was partially supported by Institute for Theoretical Computer Science, Prague (project 1M0545 of MŠMT ČR) and grant IAA100190902 of GA AV ČR. Zajíček and Ebenlendr are also partially supported by Institutional Research Plan No. AV0Z10190503.

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.*, 6:395–404, 2003.
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.*, pages 53–64. Springer, 2006.
3. D. Ye and G. Zhang. Maximizing the throughput of parallel jobs on hypercubes. *Inform. Process. Lett.*, 102:259–263, 2007.
4. O. Zajíček. A note on scheduling parallel unit jobs on hypercubes. *Int. J. on Found. Comput. Sci.*, 20(2):341–349, 2009.

Appendix

A Algorithm

Here is the promised explicit description of the algorithm that satisfies used constraints.

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 set be called *a T-schedulable set*.

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

Lemma 8. *All T-conforming sets have the same number of jobs of specific sizes.*

Proof. Let us have a 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 have to be an even number (otherwise it would not be possible to balance the sum of sizes by bigger jobs) and there have 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 8). We define *a 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 just chooses a T-preferred set at time T. The T-preferred set can be constructed efficiently as follows:

1. Sort available jobs according to job size in increasing order. In case of a tie, $\prec$-smaller jobs are preferred.
2. Choose as many jobs as possible (the sum of chosen jobs is not allowed to exceed m) in 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 9. *Set C from the algorithm 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 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. $\square$

Theorem 3. *During time T, the algorithm finds the T-preferred set.*

Proof. By Lemma 9, 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. $\square$