# A Note on Scheduling Parallel Unit Jobs on Hypercubes

Ondřej Zajíček [*]

**Abstract**

We study the problem of scheduling independent 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. Jobs' intervals of feasibility have to be nested. We provide an polynomial time algorithm for the problem.

## 1 Introduction

We study the problem of scheduling unit-time parallel jobs (also called multiprocessor tasks) on a parallel machine with hypercube topology of processor network. The jobs have to be scheduled between their release times and deadlines and the goal is to maximize the number of early jobs.

The scheduling of sequential (requiring one machine) unit jobs into time intervals on one processor in order to maximize the weighted number of early jobs is one of elemenary scheduling problems. This problem can be easily solved in polynomial time [1]. Even the generalized problem of scheduling sequential unit jobs on parallel uniform machines is polynomialy solvable [1].

The next step in the generalization of this problem is the introduction of parallel jobs. The parallel job variant of the problem requests that several processors (from the set of identical processors) are assigned to each job

[*]`ondrej.zajicek@mff.cuni.cz` Institute of Mathematics, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic. Partially supported by Institutional Research Plan No. AV0Z10190503, by ITI, Prague (project 1M0545 of MŠMT ČR) and grant IAA1019401 of GA AV ČR.

for a given time. Such variant is strongly NP-complete even in the case of identical release times and deadlines [3] because tight instances require tight packing of jobs of common sizes to each timeslot and we can reduce well-known 3-partition problem to that. We inquire which restricted problems are polynomially solvable.

Real multiprocessor machines usually have some nontrivial structure. Theoretical models for scheduling problems take this into account by assuming a specific network topology of the processors. Such topology implies restriction that jobs are allowed to be scheduled only on near processors. The structure of a multiprocessor network is often modelled using a graph, where graph vertices represent processors and graph edges represent communication lines between them. Parallel jobs have to be scheduled on the certain class of connected subgraphs. For example, in the mesh topology the set of processors forms a rectangular mesh and jobs can by scheduled on subrectangles of that mesh.

We address the scheduling problem on hypercubes, in which processors are connected to form a hypercube and jobs can be scheduled on appropriate sub-hypercubes. Such jobs have a size that is a power of two; therefore, packing such jobs into timeslots is easy. But still 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 [6] showed that the maximization problem in this model is polynomially solvable if we suppose identical release times. We will show that the maximization problem is polynomially solvable in a more general setting, in which we allow different release times, but we request that jobs' intervals of feasibility are nested. Time complexity of our algorithm is $O(n \log m)$,

An even more restricted model is the 'tall/small' model, in which only jobs that request one or all processors are allowed. Baptiste and Schieber [4] showed that feasibility testing in the 'tall/small' model is polynomially solvable. The article contains two algorithms for the 'tall/small' problem, which showed to be surprisingly difficult. Note that there is a correctable error in the proof of the first algorithm [5]. It is an open problem whether the maximization variant of this problem (denoted as $P2|r_i; p_i = 1; size_i| \sum U_i$) is polynomially solvable even for two machines [2], which is a special case of scheduling problem on hypercubes.

## 2 Preliminaries

The problem has a parameter $m$ giving the number of processors which is a power of two. An instance of the problem consists of a set of $n$ jobs. Each job $J$ has a release time $r_J$, a deadline $d_J$ and a size $s_J$ (the number of requested processors), which is also a power of two. All jobs have unit processing time, their release times and deadlines are integers and their sizes are powers of two. The objective is to find a schedule maximizing a number of processed jobs — for each processed job $J$ find a timeslot $t_J$ satisfying $r_J \leq t_J < d_J$. Naturally, the sum of sizes of jobs scheduled in one timeslot must be less or equal to $m$.

Instead of bringing a precise definition of what is a schedule consistent with a hypercube topology we use a stricter concept of *aligned schedules.* Let processors are numbered from 0 to $m-1$. We define a *block* of size $s$ (where $s$ is a power of two) as a set of processors $\{si, \ldots, (si+s-1)\}$ for an integer $i$ (satisfying $0 \leq i$ and $si + s \leq m$). We request that a job of size $s$ has to be scheduled on a block of size $s$. We use the term *aligned schedule* for a schedule satisfying this property.

We suppose a hypercube topology where two processors are connected iff their numbers in the binary notation differ in exactly one bit. In this topology a block is a subhypercube and every aligned schedule is also consistent with such topology. In this article, we will not consider non-aligned schedules to be valid schedules for this problem. This restriction is without loss of generality because if we have a set of jobs that are scheduled in one timeslot and a sum of their sizes is less or equal to $m$, we can greedily assign blocks to jobs in order from bigger jobs to smaller ones to get an aligned schedule. Therefore, the essential difference between scheduling on hypercubes and in a general case is that sizes of jobs are powers of two.

Besides this, there is another restriction compared to a general case. We suppose that intervals $[r_J, d_J)$ are nested; that is, for each two jobs $J$ and $J'$ either $[r_J, d_J) \subseteq [r_{J'}, d_{J'})$, $[r_{J'}, d_{J'}) \subseteq [r_J, d_J)$ or $[r_J, d_J) \cap [r_{J'}, d_{J'}) = \emptyset$.

For job $J$, let $I_J$ be an interval $[r_J, d_J)$. We use $r(I)$ and $d(I)$ for bounds of interval $I$. Let $INT$ be a set of intervals used in an instance of the problem. The interval $I$ is a member of $INT$ iff there is job $J$ such that $I = I_J$.

Now we define a tree structure on $INT$. For intervals $I$ and $I'$ from $INT$, $I'$ is a child of $I$ iff $I'$ is subinterval of $I$ and there is no interval $I''$ in $INT$ that is between $I$ and $I'$ (in the sense of the subinterval ordering). We can suppose without loss of generality that there is only one maximal interval in $INT$ so we get a tree and not a forest. (Otherwise we can split the problem to independent sets of jobs where each set consists of jobs that have to be scheduled somewhere in one particular maximal interval.)

In the algorithm we use packs, which are essentially sets of jobs with an additional size attribute. Each pack $P$ contains a set of jobs and has a size $s(P)$ that is a power of two and is larger or equal than the sum of sizes of jobs from pack $P$. Jobs from pack $P$ have to be scheduled together on one block of size $s(P)$ and no other job may be scheduled on this block. In many cases we do not distinguish between a job of size $s$ and a pack of size $s$ that contains just this job.

The algorithm maintains one set $S(I)$ for each vertex $I$ (from $INT$) of the tree. These sets contain packs of jobs that are decided to be scheduled during interval $I$. At the beginning, these sets are empty. During the run of the algorithm, new packs are added to these sets, moved from one set to another set or replaced by bigger packs. We define $T(I)$ as a set of jobs in packs in $S(I)$ and $U(I)$ as a union of all $T(I')$ where $I'$ is $I$ or a descendant of $I$ (union over vertices in the subtree).

For each vertex $I$ we define $q(I)$ as the sum of sizes of packs in the set $S(I)$, i.e., $q(I) = \sum_{P \in S(I)} s(P)$. We define $p(I)$ as sum of $q(I')$ for the subtree of $I$, i.e., $p(I) = \sum_{I' \in INT, I' \subseteq I} q(I')$. We define $w(I)$ as a workforce of the interval, i.e., let $I = [r, d)$, then $w(I) = (d - r)m$. We say that interval $I$ is full iff $p(I) = w(I)$ and $I$ is overfull iff $p(I) > w(I)$.

## 3 Algorithm

The algorithm runs in rounds. Jobs of the same size are processed during each round. Let $s$ be the size of jobs processed during current round. In the first round $s = 1$, then $s$ is doubled during each transition to the next round, and $s = m$ in the last round.

Algorithm maintains these invariants:

**Invariant 1** *In every round for each interval $I$ set $S(I)$ contains only packs of size $s$ and $2s$. At the beginning of a round and during the first phase, all these packs has size $s$, at the end of a round all these packs has size $2s$. Accordingly, $q(I)$ is also a multiple of $s$ (or $2s$).*

**Invariant 2** *Jobs are not assigned to inappropriate intervals; i.e., for each interval $I$ and for each job $J$ from $T(I)$, $I$ is a subinterval of $I_J$.*

**Invariant 3** *Intervals are not overfull; i.e., for each interval $I$, $p(I) \leq w(I)$.*

During each round there are two phases. During the first phase the algorithm processes every job of size $s$ in any order. For each job $J$, if $I_J$ and all its ancestors are not full, then a new pack (of size $s$) containing job $J$ is created and inserted to $S(I_J)$; otherwise, job $J$ is discarded. By invariants 1 and 3, if any interval $I$ is not full, then $p(I) + s \leq w(I)$; hence, interval $I$ will not be overfull after inserting job $J$.

During the second phase the algorithm processes every vertex in a topological order, so that every vertex is processed after all of its children. For each vertex $I$, by invariant 1, $q(I)$ is a multiple of $s$. If $q(I)$ is an odd multiple of $s$, then algorithm looks for any pack in $I'$, the nearest nonempty ($q(I') \neq 0$) ancestor of $I$, and moves that pack from $S(I')$ to $S(I)$. The size of the pack is $s$, so if there is any nonempty ancestor of $I$, the algorithm makes $q(I)$ to be an even multiple of $s$, After that, the algorithm chooses any matching of the packs in $S(I)$ and replaces each matched two packs (of size $s$) by one pack of size $2s$ containing the jobs from these two packs. If there is an odd number of packs in $S(I)$ (in the case that all ancestors of $I$ are empty and previous step does not bring one pack to $I$), then the remaining one pack is replaced by a new pack of size $2s$ containing only the jobs from the replaced pack; this will increase $q(I)$ by $s$. After this step, all packs in $S(I)$ are of size $2s$.

In the last round, instead of the second phase there is a final phase in which the algorithm processes every vertex $I$ of the tree in a topological order and for each pack in $S(I)$ (now of size $m$) it assigns any free timeslot in interval $I$. Because of invariant 3, there will be enough of free timeslots.

In Algorithm 1, there is the algorithm written in a pseudocode. The pseudocode is written with assumption that $q(I)$ and $p(I)$ are computed on the fly from a data structure holding $S(I)$. In the pseudocode we represent packs as simple sets, because the size of a pack is known from invariant 1, all sizes of packs in $S(I)$ change from $s$ to $2s$ on the line with comment STEP D, SECOND PART.

## 4 Proof

Now we prove that the algorithm finds a maximal schedule. There is a correspondence between a state of the algorithm (all sets $S(I)$, the set $D$ of discarded jobs and the structure of packs) and a set of constraints on

**Algorithm 1**

```
{ INITIALIZATION }
$D \leftarrow \emptyset$
for all $I \in INT$ do
  $S(I) \leftarrow \emptyset$
end for

for $s$ in $\{1, 2, 4, 8, \ldots, m\}$ do

  for all $J \in \{J \in JOBS \mid s_J = s\}$ do { FIRST PHASE }
    if $(\forall I \in INT, I \supseteq I_J)(p(I) < w(I))$ then { STEP A }
      $S(I_J) \leftarrow S(I_J) \cup \{\{J\}\}$
    else { STEP B }
      $D \leftarrow D \cup \{J\}$
    end if
  end for

  if $s < m$ then { SECOND PHASE }
    for all $I \in INT$, in the topological order do
      if $q(I)$ is an odd multiple of $s$ then
        let $C = \{I \in INT \mid I \supset I_J, S(I) \neq \emptyset\}$
        if $C \neq \emptyset$ then { STEP C }
          let $I' =$ the $\subseteq$-minimum from $C$
          let $P =$ any pack from $S(I')$
          $S(I') \leftarrow S(I') \setminus \{P\}$
          $S(I) \leftarrow S(I) \cup \{P\}$
        end if
      end if
      let $S_{new} = \emptyset$
      while $|S(I)| \geq 2$ do { STEP D, FIRST PART }
        let $P_1, P_2 =$ any two packs from $S(I)$
        $S(I) \leftarrow S(I) \setminus \{P_1, P_2\}$
        $S_{new} \leftarrow S_{new} \cup \{P_1 \cup P_2\}$
      end while
      $S(I) \leftarrow S_{new} \cup S(I)$ { STEP D, SECOND PART }
    end for

  else { FINAL PHASE }
    for all $I \in INT$, in the topological order; $P \in S(I)$ do
      assign any free timeslot in $I$ to $P$
    end for
  end if
end for
```

schedules. As the algorithm advances, new constraints appear and the set of schedules satisfying those constraints shrinks and finally only some maximal schedules remain. Alternatively, we can suppose that if we start with any maximal schedule, then during each step of the algorithm we can modify the schedule (without decreasing its value) to satisfy the new constraints and finally we get the same schedule as the algorithm returns.

The constraints are:

1. No job in set $D$ is scheduled.
2. For each $I$, all jobs from $T(I)$ are scheduled during $I$.
3. Jobs in each pack $P$ are scheduled in one timeslot and occupy block of $s(P)$ processors. No other jobs are scheduled in that timeslot on that block.

We use the term *good schedule* for a schedule satisfying these constraints defined by current state of the algorithm.

**Lemma 4** *During the run of the algorithm, for any interval $I$ and any good schedule there are exactly $p(I)/s$ blocks (of size $s$) of processors occupied by jobs from $U(I)$.*

**Proof:** For every interval $I'$, there are only packs of size $s$ or $2s$ in $S(I')$ with total size of $q(I')$; hence, $q(I')$ is a multiple of $s$. According to the third constraint, jobs from these packs (jobs from $T(I')$) occupy $q(I')/s$ blocks of size $s$. $p(I)$ is a sum of $q(I')$ over subtree of $I$, $U(I)$ is a union of $T(I')$ over subtree of $I$. ■

**Theorem 5** *The algorithm returns a maximal schedule.*

**Proof:** At the beginning, all valid schedules are good. We choose any maximal schedule. The algorithm does only four kinds of steps (steps A, B, C and D) that change a state of the algorithm and therefore a set of good schedules. We show that if we have some maximal schedule that was good before that step then we can change it to some schedule of the same value that is good after that step. Therefore, there is a maximal schedule that is good in the final phase.

Just before the final phase all jobs are either in set $D$ or in $T(I)$ for some $I$. Hence, all good schedules have the same set of scheduled jobs and the same value. Therefore, if any maximal schedule is a good schedule, then any good schedule is maximal and the algorithm in the final phase chooses any of them.

**Step A**: The algorithm inserts job $J$ to $S(I)$. Suppose we have some maximal schedule that is good before this step. Because $I$ is not full and according to Lemma 4, there is at least one block (of size $s$) in $I$ in the schedule that does not contain jobs from $U(I)$. If the block is empty or contains only job $K$ not yet processed, then we modify the schedule to contain job $J$ in the block instead of job $K$. The block cannot contain more jobs not yet processed, because all jobs smaller than $s$ are already processed and jobs in schedule are aligned. We inserted job $J$ and removed job $K$; hence, the resulting schedule has the same value.

If the block contains jobs from $T(I')$, $T(I'')$, ... where $I'$, $I''$, ... are some ancestors of $I$, then these jobs are smaller than $s$ and do not overlap the block. Acording to the third constraint and invariant 1, all these jobs are part of one pack $P$ (of size $s$); therefore they are from one $T(I')$ for one ancestor $I'$. We modify the schedule to contain job $J$ instead of pack $P$ and we recursively use the same argument to show that we can modify the schedule to contain pack $P$ in another block because $I'$ (like all ancestors of $I$) is also non-full. In both cases the modified schedule is good after this step.

**Step B**: The algorithm discards job $J$. Job $J$ is discarded because some interval $I$ is full. By Lemma 4, in any good schedule every block of size $s$ in interval $I$ contains some jobs from $U(I)$; hence, it cannot contain job $J$ (which is of size $s$). Job $J$ also cannot be scheduled outside of $I$, because $I$ is an ancestor of $I_J$. Therefore, any maximal schedule that is good before this step is also good after this step.

**Step C**: The algorithm moves pack $P$ from $S(I')$ to $S(I)$, $I'$ is the nearest non-empty ancestor of $I$. Suppose we have some maximal schedule that is good before this step. Because $q(I)$ is an odd multiple of $s$ (the condition of using this step) and according to Lemma 4, there are the odd number of blocks (of size $s$) in $I$ in the schedule that contain jobs from $U(I)$. Because $s < m$ (in the last round this step is not used), the total number of blocks in $I$ is even. Each unprocessed job occupies at least two blocks (hence an even number of blocks), because this step is used in the second phase when all jobs of size $s$ (and smaller) are already processed. Therefore, there is a block in $I$ that does not contain jobs from $U(I)$ nor unprocessed jobs. Such a block is empty or contains pack $P'$ of jobs from $S(I'')$, where $I''$ is an ancestor of $I'$ or $I' = I''$ (because all intervals between $I$ and $I'$ are empty). We can swap

$P$ and $P'$ in the schedule and the modified schedule has the same value and is good after this step.

**Step D**: The algorithm pairs packs of size $s$ from $S(I)$ and replaces them by packs of size $2s$. Suppose we have some maximal schedule that is good before this step. By invariant 1, there are $q(I)/s$ packs in $S(I)$ and because of the third constraint these packs occupy $q(I)/s$ blocks of size $s$ in the schedule. Let $S_1$ is the set of these blocks and $S_2$ is the set of their neighbour blocks (in the hypercube) except for blocks already in $S_1$. Blocks from $S_2$ cannot contain jobs from $T(I)$ (such blocks are in $S_1$), jobs from $T(I')$ where $I'$ is a descendant of $I$ (because packs from $S(I')$ are already packed to size $2s$; hence, each of them occupies two neighouring blocks of size $s$), unprocessed jobs (these are of size at least $2s$; hence, each of them occupies also two neighouring blocks of size $s$) and discarded jobs (by the first constraint). Therefore, blocks from $S_2$ can be free or contain packs from $S(I')$, where $I'$ is some ancestor of $I$.

Packs from $S(I)$ and from $S(I')$ (where $I'$ is any ancestor of $I$) can be moved (in schedule) to any block from $S_3 = S_1 \cup S_2$ by invariant 2. For each block of size $s$ in $S_3$, its neighbour is also in $S_3$ and so $S_3$ contains the even number of blocks. We can think of $S_3$ as it contains at least $q(I)/2s$ blocks of size $2s$. We modify the schedule by rearranging jobs on blocks from $S_3$ in such a way that we put newly created packs of size $2s$ and remaining packs (scheduled on blocks from $S_2$) to these blocks of size $2s$.

If $q(I)/s$ is even, then newly created packs have the same sum of sizes as source packs and just a rearrangement of jobs in the schedule is enough to satisfy the new constraint. If $q(I)/s$ is odd, then one pack of size $s$ is replaced by a new pack of size $2s$ (with the same jobs); therefore, we have to reserve a block of $s$ free processors. However, $q(I)/s$ is odd only when algorithm failed to bring another pack to $I$, because all ancestors of $I$ are empty. Therefore, all blocks from $S_2$ are free and there is at least one block from $S_2$ to cover the increase of the size of the the pack. The modified schedule has the same value and is good after this step. ■

A naive implementation of the algorithm (like the one in Algorithm 1, using double-linked lists for the representation of sets) has a running time of $O(n^2 \log m)$, as there are $\log m$ iterations of the outer cycle, at most $n$ iterations of the cycle over intervals in $INT$ and $O(n)$ time for tests in step A/B and C and for the cycle in D.

There are two problematic things in the algorithm – the tests in step A/B and in step C. Firstly, we examine step A/B. A reasonable representation of $S(I)$

allows us to query $q(I)$ in a constant time, but $p(I)$ is not directly accessible. Before the first phase, we compute $p(I)$ for each $I$, which can be done in time $O(n)$ if computed values for children intervals are used to compute value for a parent interval. After that, we walk over the tree structure on $INT$ and before leaving interval $I$, we process jobs $J$ such that $I_J = I$ and $s_J = s$. During the walk, we maintain a heap of intervals on a path from the current interval to the root. We also maintain variable $T$ storing a total size of jobs accepted so far during this phase. When we enter interval $I$, we put that interval to a heap with a key value $k(I) \leftarrow w(I)-p(I)+T$. Because we didn't processed yet jobs from a subtree of $I$, $p(I)$ is the value computed before this phase. For each interval $I'$ in the heap, $k(I') - T = w(I') - p(I')$. When jobs are accepted, then $p(I')$ increases as well as $T$ and the equations hold. To process job $J$, we just look at the minimum value $k_{min}$ in the heap and accept (and increase $T$) when $k_{min} > T$, otherwise we reject. When leaving interval $I$, we remove $I$ from the heap.

We can see that we use the heap in a LIFO manner. Therefore, we can replace the heap with a stack and insert interval $I$ to the stack only if $k(I) < k_{min}$. In that case, $k_{min}$ value is directly accessible on top of the stack, all operations take constant time and the complete first phase together with computing $p(I)$ takes time $O(n)$.

An inner cycle in the second phase of the algorithm can be regarded as a postorder traversal of the tree structure on $INT$. During the walk, we can maintain a stack of nonempty vertices on a path from a currently visited vertex to the root. The stack allows testing the condition in step C and finding $I'$ in time $O(1)$. Maintaining the stack during the tree walk obviously does not worsen its time complexity. The number of moves in step C over the run of the algorithm is at most the number of all iterations of the cycle in step D because every move of a pack in step C is followed by a merge of two packs in step D. The number of packs (and thus the number of moves in step C and iterations in step D) is at most $n$. Therefore, all iterations of the second phase take time $O(n \log m)$.

The final phase of the algorithm takes time $O(n)$. Therefore, such implementation of the algorithm runs in time $O(n \log m)$.

## 5 Conclusion

We addressed the scheduling problem of parallel unit jobs with nested intervals on hypercubes to maximize the number of early jobs. We have presented the algorithm for the problem running in time $O(n \log m)$. The result is a generalization of previously published result of D. Ye and G. Zhang [6].

The remaining question is whether this scheduling problem without the nested intervals restriction (or at least the optimization scheduling of tall/small jobs) is polynomially solvable. Another interesting question is whether the approach can be extended to the weighted variant of the problem.

## References

[1] P. Brucker, Scheduling Algorithms, pages 85; 124

[2] P. Brucker and S. Knust, Complexity results for scheduling problems, http://www.mathematik.uni-osnabrueck.de/research/OR/class/

[3] A. V. Fishkin and G. Zhang, On maximizing the throughput of multiprocessor tasks, *Theoretical Computer Science*, vol. 302 (2003), pp. 319–335.

[4] P. Baptiste and B. Schieber, A note on scheduling tall/small multiprocessor tasks with unit processing time to minimize maximum tardiness, *Journal of Scheduling*, vol. 6 (04/2003), pp. 395–404.

[5] P. Baptiste and B. Schieber, personal communication.

[6] D. Ye and G. Zhang, Maximizing the throughput of parallel jobs on hypercubes, *Information Processing Letters*, vol. 102 (06/2007), pp. 259–263.