

Scheduling and Traffic Allocation for Tasks with Bounded Splittability^{*}

Piotr Krysta¹, Peter Sanders¹, and Berthold Vöcking²

¹ Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken,
Germany. {krysta,sanders}@mpi-sb.mpg.de

² Dept. of Computer Science, Universität Dortmund, Baroper Str. 301, 44221
Dortmund, Germany. voecking@ls2.cs.uni-dortmund.de

Abstract. We investigate variants of the problem of scheduling tasks on uniformly related machines to minimize the makespan. In the k -splittable scheduling problem each task can be broken into at most $k \geq 2$ pieces to be assigned to different machines. In a more general SAC problem each task j has its own splittability parameter $k_j \geq 2$. These problems are NP-hard and previous research focuses mainly on approximation algorithms. Our motivation to study these scheduling problems is traffic allocation for server farms based on a variant of the Internet Domain Name Service (DNS) that uses a stochastic splitting of request streams. We show that the traffic allocation problem with standard latency functions from Queueing Theory cannot be approximated in polynomial time within any finite factor because of the extreme behavior of these functions.

Our main result is a polynomial time, exact algorithm for the k -splittable scheduling problem as well as the SAC problem with a fixed number of machines. The running time of our algorithm is exponential in the number of machines but is only linear in the number of tasks. This result is the first proof that bounded splittability reduces the complexity of scheduling as the unsplittable scheduling is known to be NP-hard already for two machines. Furthermore, since our algorithm solves the scheduling problem exactly, it also solves the traffic allocation problem.

1 Introduction

A server farm is a collection of servers delivering data to a set of clients. Large scale server farms are distributed all over the Internet and deliver various types of site content including graphics, streaming media, downloadable files, and HTML on behalf of other content providers who pay for an efficient and reliable delivery of their site data. To satisfy these requirements, one needs an advanced traffic management that takes care for the assignment of traffic streams to individual servers. Such streams can be formed, e.g., by traffic directed to the same page, traffic directed to pages of the same content provider, or by the traffic requested from clients in the same geographical region or domain, or also by combinations

^{*} Partially supported by DFG grants Vo889/1-1, Sa933/1-1, and the IST program of the EU under contract IST-1999-14186 (ALCOM-FT).

of these criteria. The objective is to distribute these streams as evenly as possible over all servers in order to ensure site availability and optimal performance.

For each traffic stream there is a corresponding stream of requests sent from the clients to the server farm. Current implementations of commercial Web server farms use the Internet Domain Name Service (DNS) to direct the requests to the server that is responsible for delivering the data of the corresponding traffic stream. The DNS can answer a query such as “What is www.uni-dortmund.de?” with a short list of IP addresses rather than only a single IP address. The original idea behind returning this list is that, in case of failures, clients can redirect their requests to alternative servers. Nowadays, slightly deviating from this idea, these lists are also used for the purpose of load balancing among replicated servers (cf., e.g., [8]). When clients make a DNS query for a name mapped to a list of addresses, the server responds with the entire list of IP addresses, rotating the ordering of addresses for each reply. As clients typically send their HTTP requests to the IP address listed first, DNS rotation distributes the requests more or less evenly among all the replicated servers in the list.

Suppose the request streams are formed by a sufficiently large number of clients such that it is reasonably well described by a Poisson process. Let λ_j denote the rate of stream j , i.e., the expected number of requests in some specified time interval. Under this assumption, rotating a list of ℓ servers corresponds to splitting stream j into ℓ substreams each of which having rate λ_j/ℓ . We propose a slightly more sophisticated stochastic splitting policy that allows for a better load balancing and additionally preserves the Poisson property of the request streams. Suppose, the DNS attaches a vector p_1^j, \dots, p_ℓ^j with $\sum_i p_i^j = 1$ to the list of each stream j . In this way, every individual request in stream j can be directed to the i th server on this list with probability p_i^j . This policy breaks Poisson stream j into ℓ Poisson streams of rate $p_1^j \lambda_j, \dots, p_\ell^j \lambda_j$, respectively.

The possibility to split streams into smaller substreams can obviously reduce the maximum latency. It is not obvious, however, whether it is easier or more difficult to find an optimal assignment if every stream is allowed to be broken into a bounded number of substreams. Observe that the allocation problem above is a variant of machine scheduling in which streams correspond to jobs and servers to machines. In the context of machine scheduling, bounded splittability has been investigated before with the motivation to speed up the execution of parallel programs. We will introduce first the relevant background in scheduling.

Scheduling on uniformly related machines. Suppose a set of jobs $[n] = \{1, \dots, n\}$ need to be scheduled on a set of machines $[m] = \{1, \dots, m\}$. Jobs are described by sizes $\lambda_1, \dots, \lambda_n \in \mathbb{Q}_{>0}$, and machines are described by their speeds $s_1, \dots, s_m \in \mathbb{Q}_{>0}$. In the classical, *unsplittable scheduling problem* on uniformly related machines, every job must be assigned to exactly one machine. This mapping can be described by an assignment matrix $(x_{ij})_{i \in [m], j \in [n]}$, where x_{ij} is an indicator variable with $x_{ij} = 1$ if job j is assigned to machine i and 0 otherwise. The objective is to minimize the *makespan* $z = \max_{i \in [m]} \sum_{j \in [n]} \lambda_j x_{ij} / s_i$. It is well known that this problem is strongly NP-hard. Hochbaum and Shmoys [5, 6] gave the first polynomial time approximation schemes (PTAS) for this problem.

If the number of machines is fixed, then the problem is only weakly NP-hard and it admits a fully polynomial time approximation scheme (FPTAS) [7].

A fractional relaxation of the problem leads to splittable scheduling. In the *fully splittable scheduling* problem the variables x_{ij} can take arbitrary real values from $[0, 1]$ subject to the constraints $\sum_{i \in [m]} x_{ij} \geq 1$, for every $j \in [n]$. This problem is trivially solvable, e.g., by assigning a piece of each job to each machine whose size is proportional to the speed of the machine.

k -splittable machine scheduling and the SAC problem. In the *k -splittable machine scheduling problem* each job can be broken into at most $k \geq 2$ pieces that must be placed on different machines, i.e., at most k of the variables $x_{ij} \in [0, 1]$, for every j , are allowed to be positive. Recently, Shachnai and Tamir [12] introduced a generalization of this problem, called *scheduling with machine allotment constraints (SAC)*. In this problem, each job j has its own splittability parameter $k_j \geq 1$. In our study, we will mostly assume $k_j \geq 2$, for every $j \in [n]$.

Shachnai and Tamir [12] prove that, in contrast to the fully splittable scheduling problem, the k -splittable machine scheduling problem is strongly NP-hard even on identical machines. They also give a PTAS for the SAC problem, whose running time, however, does not render practical as the splittability appears double exponentially in the running time. As a more practical result, they present a very fast $\max_j(1 + 1/k_j)$ -approximation algorithm. This result suggests that, in fact, approximation should get easier when the splittability is increased.

We should mention here, that there is a related scheduling problem in which *preemption* is allowed, that is, jobs can be split arbitrarily but pieces of the same job cannot be processed at the same time on different machines. Shachnai and Tamir study also combinations of SAC and scheduling with preemption in which jobs can be broken into a bounded number of pieces and additionally there are bounds on the number of pieces that can be executed at the same time. Further variants of scheduling with different notions of splittability with motivations from parallel computing and production planning can be found in [12] and [13].

Scheduling with non-linear latency functions. The only difference between the k -splittable scheduling and the traffic allocation problems is that the latency occurring at servers may not be linear. A typical example of a latency function at a server of speed s with an incoming Poisson stream at rate λ is $f_s(\lambda) = \frac{\lambda}{s(s - \min\{s, \lambda\})}$. This family of functions can be derived from the formula for the waiting time on an M/M/1 queueing system. Of course, M/M/1 waiting time is only one out of many examples for latency functions that can be obtained from Queueing Theory. In fact, a typical property of such functions is that the latency goes to infinity when the injection rate approaches the service rate.

Instead of focusing on particular latency functions, we will set up a more general framework to analyze the effects of non-linearity. The *k -splittable traffic allocation problem* is a variant of the k -splittable scheduling. Streams are described by rates $\lambda_1, \dots, \lambda_n$, and servers—by bandwidths or service rates s_1, \dots, s_m . Hence, traffic streams can be identified with jobs and servers with machines. The latencies occurring at the servers are described by a *family of latency functions*

$\mathcal{F} = \{f_s : \mathbb{Q}_{\geq 0} \rightarrow \mathbb{Q}_{\geq 0} \cup \{\infty\} \mid s \in \mathbb{Q}_{> 0}\}$ where f_s denotes a non-decreasing latency function for a server with service rate s .

Scheduling under non-linear latency functions has been considered before. Alon et al. [2] give a PTAS for makespan minimization on identical machines with certain well-behaved latency functions. This was extended into a PTAS for makespan minimization on uniformly related machines by Epstein and Sgall [4]. In both studies, the latency functions must fulfill some analytical properties like convexity and uniform continuity under a logarithmic scale. Unfortunately, the uniform continuity condition excludes typical functions from Queueing Theory.

Our results. The main result of this paper is a fixed-parameter tractable algorithm for the k -splittable scheduling and the more general SAC problem with splittability at least two for every job. Our algorithm has polynomial running time for every fixed number of machines. This result is remarkable as unsplittable scheduling is known to be NP-hard already on two machines. In fact, our result is the first proof that bounded splittability reduces the complexity of scheduling.

In more detail, given any upper bound T on the makespan of an optimal assignment, our algorithm computes a feasible assignment with makespan at most T in time $O(n + m^{m+m/(k_0-1)})$ with $k_0 = \min\{k_1, k_2, \dots, k_n\}$. Furthermore, despite the possibility to split the jobs into pieces of non-rational size, we prove that the optimal makespan can be represented by a rational number with only a polynomial number of bits. Thus the optimal makespan can be found by using binary search techniques over the rationals. This yields an exact, polynomial-time algorithm for SAC with a fixed number of machines. (We have recently improved the running time in case of identical machines [1].) Note, that this problem is strongly NP-hard when the number of machines is not fixed and $k_0 \geq 2$ [12].

In addition, we study the effects due to the non-linearity of latency functions. The algorithm above can be adopted to work efficiently for a wide class of latency functions containing even such extreme functions as M/M/1 waiting time. On the negative side, we prove that latency functions like M/M/1 do not admit polynomial time approximation algorithms with finite approximation ratio if the number of machines is unbounded. The latter result is an ultimate rationale for our approach to devise efficient algorithms for a fixed number of machines.

2 An exact algorithm for SAC with given makespan

We present here an exact algorithm for SAC with $k_j \geq 2$ for every job. Our algorithm has polynomial running time for any fixed number of machines. We assume that an upper bound on the optimal makespan is given. This upper bound defines a capacity for each machine. The capacity of machine i is denoted by c_i . The computed schedule has to satisfy $\sum_{j \in [n]} \lambda_j x_{ij} \leq c_i$, for every $i \in [m]$.

A difficult subproblem to be solved is to decide into which pieces of which size the jobs should be cut. In principle, the number of possible cuts is unbounded. We will show that it suffices to consider only those cuts that “saturate” a machine. Let $\pi_{ij} = \lambda_j x_{ij}$ denote the size of the piece of job j allocated to machine i .

Machine i is *saturated* by job j if $\pi_{ij} = c_i$. Our algorithm (Algorithm 1) schedules the *bulkiest* job j first where the bulkiness of j is $\lambda_j/(k_j - 1)$. Using backtracking it tries all ways to cut one piece from job j s.t. a machine is saturated. The saturated machine is removed from the problem; the splittability and size of j is reduced accordingly. The remaining problem is solved recursively. Two special cases arise. If j is too small to saturate k_j machines, all remaining jobs can be scheduled using a simple greedy approach known as McNaughton's rule [10]. Since the splittability k_j of a job is decreased whenever a piece is cut off, a remaining piece can eventually become unsplittable. Since this remaining piece will be infinitely bulky, it will be scheduled next. In this case, all machines that can accommodate the piece are tried. For the precise description see Fig. 1.

```

 $I := [m]; J := [n]$  -- Machines to be saturated; Jobs to be scheduled
if  $\sum_{j \in J} \lambda_j > \sum_{i \in I} c_i \vee \neg \text{solve}()$  then output "no solution possible"
else output nonzero  $\pi_{ij}$  values
Function solve() : Boolean
  if  $J = \emptyset$  then return true
  find a  $j \in J$  that maximizes  $\lambda_j/(k_j - 1)$ 
  if  $k_j = 1$  then -- Unsplittable remaining piece
    forall  $i \in I$  with  $c_i \geq \lambda_j$  do
       $\pi_{ij} := \lambda_j; c_i := c_i - \lambda_j; J := J \setminus \{j\}$  -- (*)
      if solve() then return true
      undo changes made in line (*)
  else -- Job  $j$  is splittable
    if  $\lambda_j/(k_j - 1) \leq \min \{c_i : i \in I\}$  then McNaughton(); return true
    forall  $i \in I$  with  $c_i < \lambda_j$  do
       $\pi_{ij} := c_i; \lambda_j := \lambda_j - c_i; k_j := k_j - 1; I := I \setminus \{i\}$  -- (**)
      if solve() then return true
      undo changes made in line (**)
  return false

Procedure McNaughton() -- Schedule greedily
  pick any  $i \in I$ 
  foreach  $j \in J$  do
    while  $c_i \leq \lambda_j$  do  $\pi_{ij} := c_i; \lambda_j := \lambda_j - c_i; I := I \setminus \{i\}$ ; pick any new  $i \in I$ 
     $\pi_{ij} := \lambda_j; c_i := c_i - \lambda_j$ 

```

Fig. 1. Algorithm 1: Find a schedule of n jobs with splittabilities k_j to m machines.

Theorem 1. *Algorithm 1 finds a feasible solution for SAC with a given possible makespan, provided that the splittability of each job is at least two. It can be implemented to run in time $\mathcal{O}(n + m^{m+m/(k_0-1)})$, where $k_0 = \min\{k_1, \dots, k_n\}$.*

Proof. All the necessary data structures can be initialized in time $\mathcal{O}(m + n)$ if we use a representation of the piece size matrix (π_{ij}) that only stores nonzero

entries. There can be at most m recursive calls that saturate a machine and at most $m/(k_0 - 1)$ recursive calls made for unsplittable pieces that remain after a job j was split $k_j - 1$ times. All in all, the backtrack tree considers no more than $m!m^{m/(k_0-1)}$ possibilities. The selection of the bulkiest job can be implemented to run in time $\mathcal{O}(\log m)$ independent of n : Only the m largest jobs can ever be a candidate. Hence it suffices to select these jobs initially using an $\mathcal{O}(n)$ time algorithm [3] and keep them in a priority queue data structure. Greedy scheduling using McNaughton's rule takes time $\mathcal{O}(n + m)$. Overall, we get an execution time of $\mathcal{O}(n + m + m!m^{m/(k_0-1)} \log m) = \mathcal{O}(n + m^{m+m/(k_0-1)})$.

The algorithm also produces only correct schedules. In particular, when $\lambda_j/(k_j - 1) \leq \min \{c_i : i \in I\}$ McNaughton's rule can complete the schedule because no remaining job is large enough to saturate more than $k_j - 1$ of the remaining machines. In particular, `solve()` maintains the invariant $\sum_{j \in J} \lambda_j \leq \sum_{i \in I} c_i$ and when McNaughton's rule is called, it can complete the schedule:

Lemma 1. *McNaughton's rule computes a correct schedule if $\sum_{j \in J} \lambda_j \leq \sum_{i \in I} c_i$ and $\forall i \in I, j \in J : \lambda_j/(k_j - 1) \leq c_i$.*

Proof. The only thing that can go wrong is that a job j is split more than $k_j - 1$ times, i.e., into $\geq k_j + 1$ pieces. Then, it completely fills at least $k_j - 1$ machines with capacity at least $\min_{i \in I} c_i$, contradicting $\lambda_j/(k_j - 1) \leq c_i$. ■

Now we come to the interesting part of the proof. We have to show that the search succeeds if a feasible schedule exists. We show the stronger claim that the algorithm is correct even if unsplittable jobs are present. (In this case only the above running time analysis would fail.) The proof is by induction on m . For $m = 1$ this is trivial since no splits are necessary.

Consider the case $m > 1$. If there are unsplittable jobs, they are infinitely bulky and so are scheduled first. Since all possible placements for them are tried, nothing can be missed for them. When a splittable job is bulkiest, only those splits are considered that saturate one machine. Lemma 2 shows that if there is a feasible schedule, there must also be one with this property. The recursive call leaves a problem with one machine less and the induction hypothesis is applicable. ■

Lemma 2. *If a feasible schedule exists and the bulkiest job is large enough to saturate a machine then there is a feasible schedule where the bulkiest job saturates a machine.*

Our approach to proving Lemma 2 is to show that any feasible schedule can be transformed into a feasible schedule where the bulkiest job saturates a machine. To simplify this task, we first establish a toolbox of simpler transformations. We begin with two very simple transformations that affect only two jobs and obviously maintain feasibility. See Fig. 2-(a) and 2-(b) for illustrations.

Lemma 3. *For any feasible schedule, consider two jobs p and q sharing machine i' , i.e., $\pi_{i'p} > 0$ and $\pi_{i'q} > 0$. For any machine i such that $\pi_{i'q} < \pi_{i'p}$ there is a feasible schedule where the overlapping piece of q is moved to machine i , i.e., $(\pi_{i'p}, \pi_{ip}, \pi_{i'q}, \pi_{iq}) := (\pi_{i'p} + \pi_{i'q}, \pi_{ip} - \pi_{i'q}, 0, \pi_{iq} + \pi_{i'q})$.*

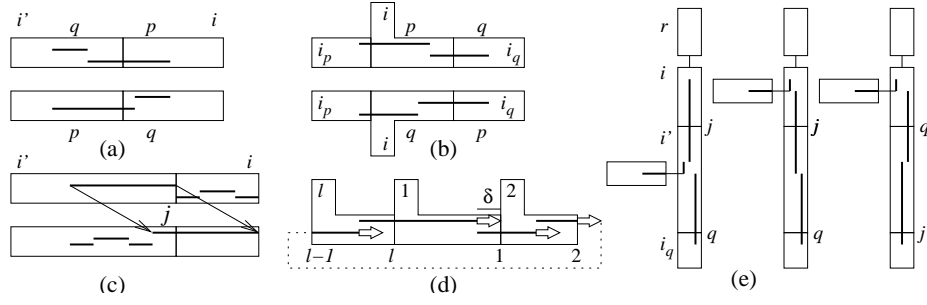


Fig. 2. Manipulating schedules. Lines represent jobs. (Bent) boxes represent machines. (a): The move from Lemma 3; (b): The swap from Lemma 4; (c): Saturation using Lemma 5; (d): The rotation from Lemma 6; (e): Moving j away from r .

Lemma 4. For any feasible schedule, consider two jobs p and q sharing machine i , i.e., $\pi_{ip} > 0$ and $\pi_{iq} > 0$. Furthermore, consider two other pieces $\pi_{i_p p}$ and $\pi_{i_q q}$ of p and q . If $\pi_{i_p p} \leq \pi_{iq} + \pi_{i_q q}$ and $\pi_{i_q q} \leq \pi_{ip} + \pi_{i_p p}$ then there is a feasible schedule where the pieces $\pi_{i_p p}$ and $\pi_{i_q q}$ are swapped as follows:

$$\begin{pmatrix} \pi_{i_p p}, & \pi_{ip}, & \pi_{i_q q}, & \pi_{i_p q}, & \pi_{iq}, & \pi_{i_q q} \end{pmatrix} := \begin{pmatrix} 0, & \pi_{ip} + \pi_{i_p p} - \pi_{i_q q}, & \pi_{i_q q} + \pi_{i_q q}, & \pi_{i_p q} + \pi_{i_p p}, & \pi_{iq} + \pi_{i_q q} - \pi_{i_p p}, & 0 \end{pmatrix}$$

As a first application of Lemma 3 we now explain how a large job j allocated to at most $k_j - 1$ machines can “take over” a small machine.

Lemma 5. Consider a job j and machine i such that $\lambda_j / (k_j - 1) \geq c_i$. If there is a feasible schedule where j is scheduled to at most $(k_j - 1)$ machines, then there is a feasible schedule where j saturates machine i .

Proof. Let i' denote a machine index that maximizes $\pi_{i'j}$ and note that $\pi_{i'j} \geq \lambda_j / (k_j - 1) \geq c_i$. We can now apply Lemma 3 to subsequently move all the pieces on machine i to machine i' . Lemma 3 remains applicable because $\pi_{i'j}$ is large enough to saturate machine i . See Fig. 2-(c) for an illustration. ■

After the above local transformations, we come to a global transformation that greatly simplifies the kind of schedules we have to consider.

Definition 1. Job j is called split if $|\{i : \pi_{ij} > 0\}| > 1$. The split graph corresponding to a schedule is an undirected hypergraph $G = ([m], E)$ where each split job j corresponds to a hyperedge $\{i : \pi_{ij} > 0\} \in E$.

Lemma 6. If a feasible schedule exists, then there is a feasible schedule whose split graph is a forest.

Proof. It suffices to show that for a feasible schedule whose split graph G contains a cycle there is also a feasible schedule whose corresponding split graph has a smaller value of $\sum_{e \in E} |e|$. Then it follows that a feasible schedule that minimizes $\sum_{e \in E} |e|$ is a forest.

So suppose G contains a cycle involving ℓ edges. Let $\text{succ}(j)$ stand for $(j + 1) \bmod \ell + 1$. By appropriately renumbering machines and jobs we can assume

without loss of generality that this cycle is made up of jobs 1 to ℓ and machines 1 to ℓ such that for $j \in [\ell]$, $\pi_{jj} > 0$, $\pi_{\text{succ}(j)j} > 0$, and $\delta = \pi_{11} = \min_{j \in [\ell]} \min \{ \pi_{jj}, \pi_{\text{succ}(j)j} \}$. Fig. 2-(d) depicts this normalized situation.

Now we *rotate* the pieces in the cycle by increasing π_{jj} by δ and decreasing $\pi_{\text{succ}(j)j}$ by the same amount. The schedule remains feasible since the load of the machines in the cycle remains unchanged. Since the first job is now split in one piece less, $\sum_{e \in E} |e|$ decreases. ■

Now we have all the necessary tools to establish Lemma 2.

Proof. Consider any feasible schedule; let j denote the bulkiest job and let there be a machine i_0 with $\lambda_j / (k_j - 1) \geq c_{i_0}$. We transform this schedule in several steps. We first apply Lemma 6 to obtain a schedule whose split graph is a forest. We now concentrate on the tree T where j is allocated. If job j is allocated to at most $k_j - 1$ machines, we can saturate i_0 using Lemma 5 and we are done.

If one piece of j is allocated to a leaf i of T then all other jobs mapped to machine i are allocated there entirely. Let i' denote another machine j is mapped to. We apply Lemma 3 to move small jobs from i to i' . When this is no longer possible, either job j saturates machine i and we are done or there is a job j' with $\lambda_{j'} = \pi_{ij'} > \pi_{i'j}$. Now we can apply Lemma 4 to pieces π_{ij} , $\pi_{i'j}$, $\pi_{ij'}$, and a zero size piece of job j' . This transformation reduces the number of pieces of job j so that we can saturate machine i_0 using Lemma 5.

Finally, j could be allocated to machines that are all interior nodes of T . We focus on the two largest pieces π_{ij} and $\pi_{i'j}$ so that $\pi_{ij} + \pi_{i'j} \geq 2\lambda_j / k_j$. Now fix a leaf r that is connected to i via a path that does not involve j as an edge. This is possible since j is connected to interior nodes only. Now we intend to move job j *away* from r , i.e., we transform the schedule such that the path between node r and job j becomes longer. (The path between a node v and a job e in a tree starts at v and uses edges $e' \neq e$ until a node is reached that has e as an incident edge.) We do this iteratively until j is incident to a leaf in T . Then we can apply the transformations described above and we are done.

We first apply Lemma 3 to move small pieces of jobs allocated to machine i' to machine i . Although this changes the shape of T , it leaves the distance between jobs j and r invariant unless j ends up in machine i' completely so that we can apply Lemma 5 and we are done. When Lemma 3 is no longer applicable, either j saturates machine i' and we are done or there is a job q with $\pi_{i'q} > \pi_{ij}$. In that case we consider the smallest other piece $\pi_{i_q q}$ of job q . More precisely, if q is split into at most $k_q - 1$ nonzero pieces we pick some i_q with $\pi_{i_q q} = 0$. Otherwise we pick $i_q = \min \{ \ell \neq i' : \pi_{\ell q} > 0 \}$. In either case $\pi_{i_q q} \leq \lambda_q / (k_q - 1)$. Recall that $\pi_{ij} + \pi_{i'j} \geq 2\lambda_j / k_j$ since this sum is invariant under the move operations we have performed. Furthermore, j is the bulkiest job so that

$$\pi_{i_q q} \leq \frac{\lambda_q}{k_q - 1} \leq \frac{\lambda_j}{k_j - 1} = \frac{2\lambda_j}{(k_j - 1) + (k_j - 1)} \leq \frac{2\lambda_j}{(k_j - 1) + 1} = \frac{2\lambda_j}{k_j} \leq \pi_{ij} + \pi_{i'j} .$$

So, we can apply Lemma 4 to pieces i and i' of job j and to pieces i' and i_q of job q . This increases the distance from job j to machine r as desired. Fig. 2-(e) gives an example where we apply Lemma 3 once and then Lemma 4. ■

3 Finding the optimal makespan

We assumed so far that an upper bound on the optimal makespan is known. The obvious idea now is to find the optimal makespan using binary search. In order to show that this search terminates one needs to prove that the optimal makespan is a rational number. This is not completely obvious as in principle jobs might be broken into pieces of non-rational size. The following lemma, however, shows that the optimal makespan can be represented by rational numbers of polynomial length. Let \mathbb{Q}_ℓ denote the set of non-negative rational numbers that can be represented by an ℓ -bit numerator and an ℓ -bit denominator and the symbol ∞ . The proof of the next lemma is omitted in this extended abstract.

Lemma 7. *There is a constant $\kappa > 0$ s.t. the value of an optimum solution to SAC problem with $k_j \geq 2$ (for all j) is in \mathbb{Q}_{N^κ} , where N is the problem size.*

By Lemma 7, the optimal makespan can be found by binary search methods over the rationals (see, e.g., [9, 11]) with Algorithm 1 as a decision oracle. Thus:

Corollary 1. *For every fixed number of machines, there is an exact polynomial time optimization algorithm for the SAC problem with splittability at least two.*

4 Solving the traffic allocation problem

In this section, we show how to apply the binary search approach to the traffic allocation problem, i.e., we solve the SAC problem with non-linear latency functions. We need to make some very modest assumptions about these functions. A latency function is *monotone* if it is positive and non-decreasing. The functions need not be continuous or strictly increasing, e.g., step functions are covered. For a monotone function $f : \mathbb{Q}_{\geq 0} \rightarrow \mathbb{Q}_{\geq 0} \cup \{\infty\}$, let the *inverse of f* be defined by $f^{-1}(y) = \sup\{\lambda \mid f(\lambda) \leq y\}$, for $y \geq f(0)$, and $f^{-1}(y) = 0$, for $y < f(0)$.

We say that a function f is *polynomially length-bounded* if for every $\lambda \in \mathbb{Q}_\ell$, $f(\lambda) \in \mathbb{Q}_{poly(\ell)}$. For example, the M/M/1 waiting time function is polynomially length-bounded although $\lim_{\lambda \rightarrow b^-} f_s(\lambda) = \infty$. This is because, for $\lambda, s \in \mathbb{Q}_\ell$ with $\lambda < s$, one can show $(s - \lambda) \in \mathbb{Q}_{2\ell}$, $s(s - \lambda) \in \mathbb{Q}_{4\ell}$ and $\lambda/(s(s - \lambda)) \in \mathbb{Q}_{8\ell}$ so that $f(\lambda) \in \mathbb{Q}_{8\ell}$. We say that a family of latency functions \mathcal{F} is *efficiently computable* if, for every $s, \lambda \in \mathbb{Q}_\ell$, $f_s(\lambda)$ and $f_s^{-1}(\lambda)$ can be calculated in time polynomial in ℓ . Observe that the functions from an efficiently computable family must also be polynomially length-bounded. It is easy to check that the M/M/1 waiting time family and other typical function families from Queueing Theory are efficiently computable. We obtain the following result whose proof is omitted.

Theorem 2. *Let \mathcal{F} be any efficiently computable family of monotone functions. Consider the SAC problem with latency functions from \mathcal{F} and splittability at least two. Suppose the best possible maximum latency can be represented by a number from \mathbb{Q}_ℓ . Then an optimal solution can be found in time $\mathcal{O}(poly(\ell) \cdot (n + m^{m+m/(k_0-1)}))$ with $k_0 = \min\{k_j : j = 1, 2, \dots, n\}$.*

Note that ℓ is an obvious lower bound on the running time of any algorithm computing the exact, optimal makespan. It is unclear if there exist latency functions s.t. ℓ cannot be bounded polynomially in the input length. If an appropriate upper bound on ℓ is not known in advance, we can use the geometric search, which can be stopped after computing the optimal latency with desired precision.

5 Non-approximability for non-linear scheduling

The M/M/1 waiting time cost function family defined in Section 1 has an infinity pole as $\lambda \rightarrow b^-$. Intuitively, this pole reflects the capacity restriction on the servers and it is typical also for other families that can be derived from Queueing Theory. The following theorem, whose proof is omitted, shows that the non-linear k -splittable scheduling even with identical servers is completely inapproximable.

Theorem 3. *Let \mathcal{F} be an efficiently computable family of monotone latency functions. Suppose there is $s \in \mathbb{Q}_{>0}$ s.t. $\lim_{\lambda \rightarrow s} f_s(\lambda) = \infty$. Then there does not exist a polynomial time approximation algorithm with finite approximation ratio for the non-linear k -splittable scheduling problem under \mathcal{F} , provided $P \neq NP$.*

References

1. A. Agarwal, T. Agarwal, S. Chopra, A. Feldmann, N. Kammenhuber, P. Krysta and B. Vöcking. An Experimental Study of k -Splittable Scheduling for DNS-Based Traffic Allocation. To appear in *Proc. of the 9th EUROPAR*, 2003.
2. N. Alon, Y. Azar, G. J. Woeginger and T. Yadid. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1:55–66, 1998.
3. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Science*, 7(4):448–461, August 1973.
4. L. Epstein and J. Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. *Proc. of the 7th ESA*, 151–162, 1999.
5. D. S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *J. ACM*, 34: 144–162, 1987.
6. D. S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17: 539–551, 1988.
7. E. Horowitz and S. K. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23:317–327, 1976.
8. J. F. Kurose and K. W. Ross. *Computer networking: a top-down approach featuring the Internet*. Addison-Wesley, 2001.
9. St. Kwek and K. Mehlhorn. Optimal search for rationals. *Information Processing Letters*, 86:23 - 26, 2003.
10. R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.
11. C. H. Papadimitriou. Efficient search for rationals. *Information Processing Letters*, 8:1–4, 1979.
12. H. Shachnai and T. Tamir. Multiprocessor Scheduling with Machine Allotment and Parallelism Constraints. *Algorithmica*, 32(4): 651–678, 2002.
13. W. Xing and J. Zhang. Parallel machine scheduling with splitting jobs. *Discrete Applied Mathematics*, 103: 259–269, 2000.