

# Fast Distributed Scheduling via Primal-Dual

Alessandro Panconesi\*

Mauro Sozio†

## ABSTRACT

In this paper we give an efficient distributed algorithm computing approximate solutions to a very general, and classical, scheduling problem. The approximation guarantee is within a constant factor of the optimum. By “efficient”, we mean that the number of communication rounds is poly-logarithmic in the size of the input. In the problem, we have a bipartite graph with computing agents on one side and resources on the other. Agents that share a resource can communicate in one time step. Each agent has a list of jobs, each with its own length and profit, to be executed on a neighbouring resource within a given time-window. Resources can execute non preemptively only one job at a time. The goal is to maximize the profit of the jobs that are scheduled. It is well known that this problem is NP-hard. A very interesting feature of our algorithm is that it is derived in a systematic manner from a primal-dual algorithm.

## Categories and Subject Descriptors

F.2 [Analysis of Algorithms and Problem Complexity]: General;  
C.2.4 [Distributed Systems]: Distributed applications

## General Terms

Algorithms, Theory

## Keywords

Distributed Algorithms, Primal-Dual, Scheduling, Peer-to-Peer

## 1. INTRODUCTION

In this paper we give efficient distributed algorithms for a very general, and classical, scheduling problem. To our knowledge this is the first distributed solution given for it. An interesting feature of our algorithms is that they are based on the primal-dual schema, a very powerful technique for algorithmic design. Our work provides additional evidence that this method might become a source of many sophisticated distributed algorithms for hard combinatorial problems [9].

The problem we study can be informally defined as follows (the technical definition is deferred to § 2). We have a bipartite graph with processors and resources on the two sides of the bipartition, respectively. Each processor has a list of jobs to be executed. Each job comes with length (processing time) and profit, as well as with one or more time windows within which it is to be executed. In fact, both profit and length may vary according to the time and in which resource the job is scheduled. A processor can schedule its jobs only on the resources that are adjacent to it in the graph. All resources are identical in the sense that a job can be scheduled on any of the neighbouring resources. The basic constraint is that a resource can process at most one job at any given time.

Processors that share a resource can communicate in one time step. The network is synchronous and message-passing: a process can communicate in one time step with all other processors with which it shares a resource. The running time of a protocol is given by the number of communication rounds. This model is universally used in the context of distributed graph algorithms. In realistic applications more refined time-accounting schemes would be required. Nevertheless, it provides a quite useful, albeit rough, approximation of the real costs incurred by a distributed algorithm.

The goal is to schedule a set of jobs in order to maximize the aggregate profit.

This is an adaptation in a distributed setting of the (maximization version of the) identical parallel machines problem, known in the literature as  $P|r_j|\sum w_j U_j$ . The variant where the length of a job may depend on the machine where it is scheduled (known as  $R|r_j|\sum w_j U_j$ ), may also be solved using our approach, provided that the length of any message (sent by processors) is polynomial in the size of the input.

The scheduling problem we study in this paper is very general and can model a large variety of situations in a distributed setting. For instance the resources could be internet hot spots for whose access several wireless devices are competing, they could be shared resources in a peer-to-peer network, or they could be servers providing web services (such as video on demand). Competing for the same resource may naturally induce a proximity relationship. For example, in a scenario of wireless devices competing for, say, hot spots, it is natural to assume that processors physically close to

\*Informatica, Sapienza University of Rome, Via Salaria 113, 00198, Roma, Italy Email: ale@di.uniroma1.it.

†Max-Planck-Institut für Informatik, Databases and Information Systems, Campus E1 4, 66123 Saarbrücken, Germany. Email: msozio@mpi-inf.mpg.de. This work was done while at the Dipartimento di Informatica of “La Sapienza” University of Rome.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA’08, June 14–16, 2008, Munich, Germany.

Copyright 2008 ACM 978-1-59593-973-9/08/06 ...\$5.00.

some resource may compete for that resource; moreover, since they are physically close, it is also natural to assume that they are able to communicate in relatively short time. Or, in a large peer-to-peer network, peers might be able to discover that they are competing for the same resource. Peers competing for the same resource can contact each other directly to run a distributed conflict resolution protocol, such as the one we present. In this paper however we are mostly interested in the theoretical implications, related to the potential of the primal-dual schema as a technique to derive efficient distributed algorithms.

We require our algorithms to be simple, in the sense that they can be simulated sequentially in polynomial-time. This implies that local computations are constrained to be polynomial time. We also require them to be efficient, in the sense that the number of communication rounds should be sub-linear, possibly poly-logarithmic in the input size. It is well-known that our scheduling problem is NP-hard. Therefore the best we can realistically hope for is to give approximated solutions. To put things in perspective recall that the best sequential approximation for this problem is within a factor of  $1/2$  [10]. In this paper we give a distributed algorithm such that, for any  $\epsilon > 0$ , it computes a  $\left(\frac{1}{20+\epsilon}\right)$ -approximation within  $O\left(\frac{T}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many rounds, where  $P_{\max}$  and  $P_{\min}$  are, respectively, the maximum and minimum profits,  $L_{\max}$  and  $L_{\min}$  are respectively, the maximum and the minimum length of any job and  $T$  is the time needed to compute a MIS in a special conflict graph induced by the input. If we denote the number of processors in the underlying bipartite network with  $n$ , the running time becomes:

- $O\left(\frac{\log N}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many communication rounds for general graphs, using the randomized MIS algorithm of Luby [6] or  $O\left(\frac{\log^2 n}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many rounds using a randomized subroutine to compute a network decomposition of the underlying bipartite communication [5]. In both cases the algorithm becomes randomized.
- $O\left(\frac{n^{\epsilon(n)}}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many communication rounds for general graphs, where  $\epsilon(n) = O(1/\sqrt{\log n})$ , using the network decomposition of [8]. In this case the algorithm is deterministic.
- $O\left(\frac{\log^2 n}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many communication rounds for bounded-growth graphs, using the network decomposition of [4]. In this case the algorithm becomes deterministic. Bounded-growth graphs include graphs commonly arising in wireless and internet applications.

In general, these time bounds and the size of messages are pseudo-polylogarithmic. They become fully polylogarithmic if  $P_{\max}$ ,  $L_{\max}$  and  $e_{\max}$  are bounded by a polynomial in  $n$ .

Besides the interest related to such a general scheduling problem, we believe that our work is interesting from the point of view of the theory of distributed computing. Our algorithms are derived from the sequential primal-dual algorithm of [10]. Primal-dual formulations in general exhibit a certain degree of parallelism and of locality that one can hope to exploit for a distributed implementation. This approach was originally introduced in [2] for the much simpler vertex cover problem. Other applications of this technique may be found in [1] where the facility location problem and vertex cover with soft capacities (a variant of vertex cover) are considered.

Recently it was also applied with success to vertex cover with capacities, a much more complex scenario combinatorially [9]. Our work further illustrates the potential of this approach. The problem we study has a combinatorial structure that is very different from vertex covering problems and it is moreover a *maximization* (as opposed to minimization) problem, a fact that carries with it several complications even in the sequential case. As it is well-known the primal-dual schema is a very powerful methodology for deriving algorithmic solutions, whose scope in the context of distributed algorithms remains to be explored. In this paper we extend it non-trivially to a hard and important combinatorial problem, very different from previous applications, and this is precious evidence that this line of research is quite fruitful.

As mentioned, we build primarily on the primal-dual algorithm in [10] to design an efficient distributed algorithm for this problem. In order to do this we need to solve several problems along the way. The main ingredient is to replace their stack with a *distributed stack* and manage it efficiently, i.e. ensure that it has polylogarithmic depth.

**Related work:** Our problem may be formulated as a positive linear programming (see Section 3) and in particular it is a packing problem. The work in [11], is one of the first concerned in solving positive linear programs in a parallel setting. Later Kuhn et al. considered the same problem in a distributed setting [12], giving a distributed  $O(\log \Delta)$ -approximation algorithm for covering problems as well as a  $O(\Delta)$  approximation algorithm for packing problems, both with polylogarithmic running time. In view of these results, our main contribution is to give a much better approximation guarantee than the one given for the general packing problem. Apart from already cited work, other interesting recent contributions making use of LP-methods in a distributed setting are [3, 4].

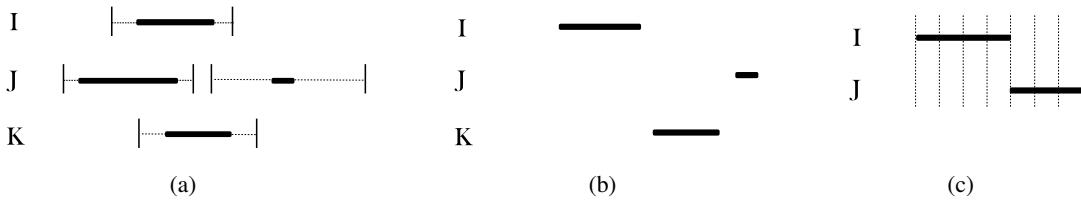
## 2. PRELIMINARIES

We will be dealing with the following scheduling problem. We are given a bipartite graph  $G = (\mathcal{P}, \mathcal{R}, E)$  where  $\mathcal{P}$  is a set of processors and  $\mathcal{R}$  is a set of resources. There is an edge between a processor  $p$  and a resource  $r$  if  $r$  is available to processor  $p$ . Each processor has a set of jobs that can be scheduled (only) in the resources available to it.

Each job comes with length (processing time) and profit, as well as with one or more time windows within which it must be scheduled. In fact, both profit and length may vary according to the time and resource where the job is scheduled. Jobs must be scheduled non-preemptively that is, once a job is scheduled it cannot be interrupted and resumed later. Figure 1a shows three jobs together with their corresponding time windows, as well as a feasible schedule of such jobs (Figure 1b) (there is only one resource).

For convenience of notation, we represent a job  $j$  with the set  $\mathcal{A}_j$  of all possible ways to schedule  $j$  within its time windows in the available resources. Each element  $I$  in  $\mathcal{A}_j$ , will be referred as an *instance* of job  $j$ , formally defined by a resource  $r$  where the job is scheduled, an interval of time with start-time  $s(I)$  and end-time  $e(I)$ , as well as a profit  $p(I)$ . We also denote with  $\mathcal{A}_j(r)$  the set of all possible instances of job  $j$  in resource  $r$ ; sets  $\mathcal{A}_j(r)$  form a partition of  $\mathcal{A}_j$ . The  $\mathcal{A}_j$ 's contain all the information we need about the jobs and from now on we shall use this representation.

In this terminology, the basic constraints become the following. For every job we can schedule at most one instance (we cannot schedule the same job twice), and any two instances belonging to different jobs cannot be on the same resource if their time intervals overlap (every resource can process at most one job at any given time). We wish to maximize the total profit of scheduled instances.



**Figure 1: (a) Jobs  $I, J$  and  $K$  together with their corresponding time windows; (b) a feasible schedule of  $I, J$  and  $K$ ; (c) two instances sharing one single point.**

In a distributed setting, only processors which share at least one resource can communicate directly (perhaps in two hops in the physical network via the common resource which might be itself a unit with computing and communication capabilities). More formally,  $p_1$  can communicate in one step with  $p_2$  if there exists a resource  $r$  such that  $p_1 r$  and  $p_2 r$  are both edges in the bipartite graph. Note that the communication graph, so defined, may have linear diameter.

In what follows we make several simplifying assumptions for sake of clarity. We will assume that time is discrete, which implies that the number of possible ways a job can be handled is finite. Furthermore we will assume that, two instances  $I$  and  $J$  such that  $e(I) = s(J)$  overlap, i.e. they overlap even if they share a single point (see Figure 1c). This clearly can be assumed without loss of generality.

### 3. IP AND LP FORMULATIONS

We start by adapting the linear program formulation in [10] to our settings. Recall that each job  $A_j$  is naturally partitioned into sets  $A_j(r)$ , where  $r$  is a neighbouring resource. Each set  $A_j(r)$  contains all the possible ways to schedule job  $A_j$  in resource  $r$ . For each instance  $I$  there is associated a start-time  $s(I)$  and an end-time  $e(I)$ . We also define a “middle-time”  $m(I)$  to be equal to  $(e(I) + s(I))/2$ . We introduce a 0 – 1 variable for each instance  $I$ , indicating whether  $I$  is scheduled or not ( $x_I = 1$  means that  $I$  is scheduled).

Let  $\mathcal{T}$  be the set of all start-times, end-times and middle-times of all instances belonging to all jobs, and let

$$\mathcal{T}_I := \{t \in \mathcal{T} | s(I) \leq t \leq e(I)\}$$

for all instances  $I$ . Furthermore, let  $\mathcal{N}$  be the set of all job indices and let  $p(I)$  denote the profit of instance  $I$ . For this scheduling problem, we obtain the following integer program formulation:

$$\max \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}} p(I) x_I \quad (\text{IP})$$

$$\text{s.t.} \sum_{j \in \mathcal{N}} \sum_{I \in A_j(r): t \in \mathcal{T}_I} x_I \leq 1 \quad \forall t \in \mathcal{T}, r \in \mathcal{R}, \quad (1)$$

$$\sum_{I \in A_j} x_I \leq 1 \quad \forall j \in \mathcal{N}, \quad (2)$$

$$x_I \in \{0, 1\} \quad \forall j \in \mathcal{N}, r \in \mathcal{R}, I \in A_j(r). \quad (3)$$

where constraint (1) says that in each resource only one job can be scheduled at any given time. Constraint (2) says that for each job, at most one instance of that job can be scheduled.

Let (LP) be the linear program obtained by replacing (3) with non-negativity constraints for these variables. Let  $u_j$  be the dual variables corresponding to the constraints (2), and let  $v_t^r$  denote the dual variables corresponding to the constraints (1), then we can

write the dual of (LP) as:

$$\min \sum_{j \in \mathcal{N}} u_j + \sum_{\substack{r \in \mathcal{R} \\ t \in \mathcal{T}}} v_t^r \quad (\text{DP})$$

$$\text{s.t.} \quad u_j + \sum_{t \in \mathcal{T}_I} v_t^r \geq p(I) \quad \forall j \in \mathcal{N}, r \in \mathcal{R}, I \in A_j(r), \quad (4)$$

$$u_j, v_t^r \geq 0 \quad \forall j \in \mathcal{N}, t \in \mathcal{T}, r \in \mathcal{R}. \quad (5)$$

Notice that in the dual we have a constraint for every instance. Instances of the same job  $A_j$  share the variable  $u_j$ , while instances in different sub-jobs  $A_j(r)$  and  $A_k(r)$  insisting on the same resource  $r$  share the “time variables”  $v_t^r$ ’s if they overlap. Thus we can use these variables for global coordination.

### 4. THE DISTRIBUTED ALGORITHM

In this section we describe our distributed algorithm. The basic idea is to parallelize the sequential primal-dual algorithm presented in [10]. To motivate the definitions to follow a brief review of that algorithm is in order. The algorithm is based on the interplay between the classical primal-dual mechanism to manipulate the dual variables, and a stack. In the beginning all dual variables are set to zero. Instances are sorted by increasing end times and are pushed in this order on the stack. For every instance there is a dual constraint, and a set of dual variables associated with the instance. When an instance  $I$  is pushed on the stack these dual variables are raised in order to saturate the dual constraint associated with the instance. This affects all constraints corresponding to instances in conflict with  $I$ . Instances whose dual constraint is satisfied are eliminated. The process continues with the remaining instances (corresponding to unsatisfied constraints), until every constraint is satisfied. At that point every instance is either in the stack or was eliminated. Since all constraints are satisfied the solution is dual feasible. The algorithm then starts a reverse process by popping instances out of the stack. When an instance is popped it is added to the primal solution (i.e. the scheduling) unless it has a conflict with previously scheduled instance, in which case it is eliminated. This ensures that the solution is feasible. The analysis shows that the scheduling so constructed is also 1/2-approximated.

We will try to mimic this by introducing one *local stack* for each processor, each one dedicated to the instances of that processor. The union of all such stacks may be seen as a unique *distributed stack*, where many instances may be pushed on (or popped out) simultaneously. Instances pushed on the stack at the same time are said to belong to the same *layer*. According to this terminology, pushing a layer on the distributed stack corresponds actually to push each instance (in the layer) on the local stacks of the corresponding processors.

Throughout our algorithm, each processor communicates only with its immediate neighbors, which implies that each processor will have a limited knowledge of the distributed stack (namely the portion corresponding to the local stack of his neighbors). In what

follows, for presentation clarity we will only deal with the distributed stack but meaning each time a set of pop and push operations in the local stacks.

One of the main problems to solve with this approach is to ensure that the depth of the stack, which in general can be linear in the input size, be bounded by a polylogarithmic function in the size of the input, for the number of steps is at least the number of push and pop operations we perform. We will see how to do this.

A second, rather technical problem is illustrated by the following example. Our input consists of two overlapping instances  $I$  and  $J$  with  $p(I) = 1$ ,  $p(J) = \epsilon > 0$ ,  $s(I) = s(J) = 0$ ,  $e(I) = 2$  and  $e(J) = 1$ . Suppose  $I$  is pushed on the stack first and suppose that the algorithm increases only the dual variable  $v_{e(I)}$ . Note that, such a variable does not appear in the constraint for  $J$ . After that,  $J$  is pushed on the stack (since his constraint is not satisfied yet) and eventually “eliminates”  $I$  in the pop phase. It is clear that it is crucial to avoid this to ensure the approximation guarantee. One way to overcome such problem is to ensure that the increasing variable  $v_{e(I)}$  appears in the constraint for  $J$ . The sequential algorithm guarantees this by ordering instances by non-decreasing end time. Since we cannot order in sub-linear time we should solve this problem in a different way.

For this reason there are 4 dual variables associated to each instance  $I$  (in contrast with the 2 dual variables in the sequential algorithm) which handles some of the problematic cases. Unfortunately, this is not sufficient as a situation like the one depicted in Figure 2 may still happen. This motivates the next design choice of our algorithm. The algorithm will execute a sequence of phases. During phase  $i$ ,  $i = 1, 2, \dots, k$ , where  $k = O(\log \frac{L_{\max}}{L_{\min}})$  only instances whose length is in the range  $(2^{i-1}, 2^i]$  are processed. At the end of phase  $i$  there will be no more instances of length  $\leq 2^i$  to be processed. This solves the problem mentioned above, as we clarify in Observation 1.

**Notation 1** Let  $I \in A_j(r)$ . In the rest of the paper, we shall refer to the four dual variables  $u_j$ ,  $v_{s(I)}^r$ ,  $v_{m(I)}^r$ ,  $e_{s(I)}^r$  as the increasing variables of instance  $I$ .

Note that for each instance  $I$ , the set of increasing variables may be a proper subset of the set of variables corresponding to  $T_I$ . Unfortunately, having four increasing variables for each instance worsens the approximation guarantee but it seems to be unavoidable.

Before we can describe what happens within a phase we need to define two more auxiliary notions. The *conflict graph* of phase  $i$  is the graph whose vertices are the instances of phase  $i$ . Two instances are connected by an edge if they are incompatible, in the following sense:

- Two instances that belong to the same job are incompatible (because they are two different ways to schedule the same job).
- Two instances  $I \in A_i(r)$  and  $J \in A_j(r)$ ,  $i \neq j$  are incompatible if  $T_I \cap T_J \neq \emptyset$ , i.e. if they overlap (because a resource  $r$  can process at most one job at any given time).

The following definition is crucial to our analysis.

**Definition 2 (Uncovered and weakly covered instances)**

Given  $\epsilon > 0$ , at any step of our algorithm we say that an instance  $I \in A_j(r)$  is uncovered if constraint (4) for  $I$  is such that

$$\bar{u}_j + \sum_{t \in T_I} \bar{v}_t^r < \frac{1}{5 + 4\epsilon} p(I) \quad (6)$$

where  $\bar{u}$  and  $\bar{v}$  denote the current value of  $u$  and  $v$ , respectively. We refer to an instance which is not uncovered as weakly covered.

In the algorithm we maintain the following invariant: at the end of phase  $i$ , no instance with length smaller than  $2^i$  is left uncovered. Said differently, at the end of phase  $i$  all instances with length less than or equal to  $2^i$  are weakly covered.

We can now finally discuss the main operations within each phase  $i$ . An instance with length within  $(2^{i-1}, 2^i]$  and such that it is yet uncovered is called an *active* instance. The following three steps are iterated until there are no active instances:

1. Compute the conflict graph induced by active instances;
2. Compute a maximal independent set in the conflict graph;
3. All instances belonging to such an independent set are pushed simultaneously on the distributed stack.
4. For each such instance  $I$ , we raise its four increasing variables by the same amount  $\delta(I)$ , in such a way the constraint (4) corresponding to instance  $I$  becomes tight.

Except from the MIS computation, these steps can be implemented in constantly many rounds in our model. The details are omitted from this extended abstract.

Note that initially all instances are uncovered but as the algorithm proceeds they may become weakly covered. This happens even if they are not pushed on the stack since they might have dual variables in common with other jobs that are pushed on the stack.

The above procedure inserts in the stack a set of layers, each layer consisting of a MIS of instances. A lemma to follow will show that the number of layers is always polylogarithmic in the input instance, a consequence of the careful choice of parameters in Definition 2. In turn, this will ensure that the running time be polylogarithmic.

When all instances, of all lengths, are weakly covered, we compute a feasible schedule in the following way. The last layer of instances is popped from the distributed stack. For every instance we check if it can be scheduled without conflicts with previously scheduled instances (when doing this we check all scheduled instances, including those scheduled in previous phases). If there are no conflicts we schedule it, otherwise we eliminate it from further consideration. We then continue with the next layer in the same fashion, and so on. We iterate until the stack becomes empty. This ends the description of the algorithm. See Algorithm 1 for the pseudocode.

To summarize, the algorithm consists of two main consecutive stages, the pushing stage followed by the popping stage. The pushing stage in turn consists of two nested loops. The outer loop is iterated for  $O(\log \frac{L_{\max}}{L_{\min}})$  phases. The inner loop consists of a sequence of MIS computations followed by local operations to raise the increasing variables associated with the instances that are pushed on the stack. We will see in a lemma to follow that the number of MIS layers is always  $O(\frac{1}{\epsilon} \log \frac{P_{\max}}{P_{\min}})$ . Therefore, the inner loop can be implemented in our message passing model within  $O(\frac{T}{\epsilon} \log \frac{P_{\max}}{P_{\min}})$  communication rounds, where  $T$  is the time complexity of the MIS procedure.

The popping stage processes the MIS layers in the distributed stack in reverse order, scheduling all instances in the layer that do not have conflicts with previously scheduled instances.

The total running time is  $O(\frac{T}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}})$  many communication rounds.

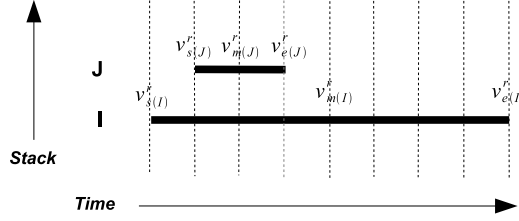


Figure 2: Instance  $J$  eliminates  $I$  in the pop phase even if its profit is arbitrarily small.

---

**Algorithm 1** The distributed algorithm

---

```

1: /* Pushing Stage */
2: for ( $i = \lfloor \log L_{\min} \rfloor$  to  $\lceil \log L_{\max} \rceil$ ) do
3:   Let  $A$  be the set of all active instance, i.e.
    $A := \{I : I \text{ uncovered} \wedge |I| \leq 2^{i+1}\}$ 
4:   while  $A$  is non empty do
5:     Let  $G[A]$  be the conflict graph induced by  $A$ .
6:     Compute a MIS  $S$  in  $G[A]$ 
7:     Push all instances of  $S$  on the distributed stack
8:     for all  $I \in S$  in parallel do
9:       Let  $j$  and  $r$  be such that  $I \in A_j(r)$ ;
10:      Let  $\delta_I = (p(I) - u_j - \sum_{t \in \mathcal{T}_I} v_t^r)/4$ ;
11:      increase  $v_{s(I)}^r, v_{m(I)}^r, v_{e(I)}^r$  and  $u_j$  by  $\delta(I)$ ;
12:     end for
13:     Update  $A$  by eliminating all instances that have become weakly
     covered
14:   end while
15: end for
16: /* Popping Stage */
17: while the distributed stack is nonempty do
18:   Pop a layer out of the distributed stack. Let this layer be  $\mathcal{L}$ 
19:   In parallel, for each instance  $I \in \mathcal{L}$ , schedule  $I$  if it has no conflicts
   with previously scheduled instances
20: end while

```

---

## 5. ANALYSIS

We now analyze the proposed algorithm. The following observations will be useful in the proof.

**Observation 1** *Let  $I$  and  $J$  be two instances such that  $I$  is pushed on the stack before  $J$ . If they are incompatible then they share at least one of  $I$ 's increasing variables.*

Observation 1 is straightforward in the case  $I$  and  $J$  belong to the same job. If  $I$  and  $J$  belong to the same phase then we have two cases to consider. If one does not contain the other then they share either  $v_{s(I)}^r$  or  $v_{e(I)}^r$ . Otherwise they will share  $v_{m(I)}^r$ . Otherwise, if they belong to different phases they will share  $v_{s(I)}^r$  or  $v_{e(I)}^r$ . See Figure 3.

**Observation 2** *The solution computed by the algorithm may not be dual feasible. However if we multiply the final value of every dual variable by  $(5 + 4\epsilon)$  the new solution is dual feasible.*

The observation follows from the fact that the algorithm ensures that every instance is weakly covered at the end of the algorithm. The next lemma establishes the approximation guarantee of the algorithm.

**Lemma 1** *For any given  $\epsilon > 0$ , the total profit of instances scheduled by Algorithm 1 is at least a fraction  $\left(\frac{1}{20+\epsilon}\right)$  of the optimum profit.*

**PROOF.** For a primal variable  $x$ , we denote with  $\bar{x}$  its value in the feasible primal solution obtained by setting  $x_I = 1$  if  $I$  is scheduled by our algorithm and 0 otherwise. For each instance  $I$  scheduled by the algorithm, we define the set  $\mathcal{C}(I)$  to be the set of all instances, including  $I$ , that are incompatible with  $I$  and that were pushed on the stack before  $I$  (during any phase). Let  $I \in A_j(r)$ .

Recall that when  $I$  is pushed on the stack its four increasing variables are raised by the same quantum  $\delta(I)$ . The intuition of the proof is as follows. When an instance  $I$  is scheduled it “eliminates” all instances incompatible with it. These eliminated instances were pushed on the stack before  $I$ . We would like to argue that the total profit of the instances eliminated by  $I$  is at most a constant fraction of the profit gained by including  $I$ . This is done by charging the deltas of the eliminated instances on the eliminator. This is possible because any eliminated instance shares an increasing variable with the eliminator.

For a dual variable  $x$  let  $\hat{x}$  denote its value when  $I$  is pushed on the stack. Let  $I \in A_j(r)$ . The following holds,

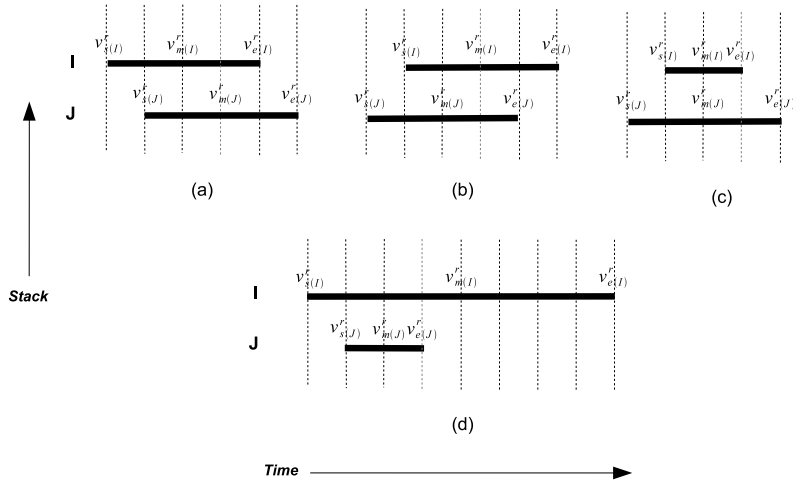
$$\sum_{J \in \mathcal{C}(I)} \delta(J) \leq \hat{u}_j + \sum_{t \in \mathcal{T}_I} \hat{v}_t^r \leq p(I). \quad (7)$$

The first inequality follows from Observation 1, while the last inequality follows from the fact that  $I$  is still *uncovered* at the time it is pushed on the stack and after that it increases its dual variables in order for constraint (4) to become tight.

For every instance  $I$  there exists  $J$  such that  $I \in \mathcal{C}(J)$  (possibly  $I = J$ ). This is because, either  $I$  is scheduled at the end of the algorithm or there is an incompatible instance  $J$  that has been scheduled instead. It follows that,

$$\begin{aligned} \sum_{\substack{j \in \mathcal{N} \\ r \in \mathcal{R}}} \sum_{I \in A_j(r)} \delta(I) &\leq \sum_{\substack{j \in \mathcal{N} \\ r \in \mathcal{R}}} \sum_{\substack{J \in A_j(r) \\ \bar{x}_j = 1}} \sum_{I \in \mathcal{C}(J)} \delta(I) \\ &\leq \sum_{\substack{j \in \mathcal{N} \\ r \in \mathcal{R}}} \sum_{\bar{x}_j = 1} p(J) \end{aligned} \quad (8)$$

The initial value of all dual variables is zero and every instance increases the total value of the dual variables by at most  $4\delta$ . From this, it follows that we can bound the final aggregate value of the dual variables with the sum over all delta's:



**Figure 3: Possible conflicts between instances processed in a same phase (a,b,c) and between instances processed in different phases (d).**

$$\begin{aligned}
\sum_{j \in \mathcal{N}} \bar{u}_j + \sum_{\substack{r \in \mathcal{R} \\ t \in \mathcal{T}}} \bar{v}_t^r &\leq 4 \left( \sum_{j \in \mathcal{N}} \sum_{\substack{r \in \mathcal{R} \\ I \in A_j(r)}} \delta(I) \right) \\
&\leq 4 \left( \sum_{j \in \mathcal{N}} \sum_{\substack{r \in \mathcal{R} \\ J \in A_j(r)}} p(J) \bar{x}_J \right) \quad (9)
\end{aligned}$$

As remarked, the solution computed by the distributed algorithm may not be feasible. Consequently the above bound does not give an approximation guarantee. However, from Observation 2 and from the fact that any feasible dual solution is an upper bound on the optimum it follows that,

$$\begin{aligned}
OPT &\leq (5 + 4\epsilon) \left( \sum_{j \in \mathcal{N}} \bar{u}_j + \sum_{\substack{r \in \mathcal{R} \\ t \in \mathcal{T}}} \bar{v}_t^r \right) \\
&\leq (20 + 16\epsilon) \left( \sum_{j \in \mathcal{N}} \sum_{\substack{r \in \mathcal{R} \\ J \in A_j(r)}} p(J) \bar{x}_J \right) \quad (10)
\end{aligned}$$

where last inequality follows from Equation (9). The claim follows, since the quantity within parenthesis in last term is the value of the solution computed by the algorithm. ■

The next two lemmas show that the inner loop of the push stage requires polylogarithmically many communication rounds. The main idea is to show that the number of MIS layers computed by the inner loop is polylogarithmic.

**Lemma 2** *Let \$I\$ and \$J\$ be two incompatible instances such that \$I\$ was pushed on the stack before \$J\$. Then,*

$$p(J) \geq (1 + \epsilon)p(I).$$

**PROOF.** At the time it is pushed on the stack, instance \$I\$ is *uncovered* and hence its increasing variables are raised by at least

$$\delta(I) \geq \left(1 - \frac{1}{5 + 4\epsilon}\right) \frac{p(I)}{4} = \frac{1 + \epsilon}{5 + 4\epsilon} p(I).$$

Since \$I\$ and \$J\$ are incompatible, it follows from Observation 1 that they share at least one of \$I\$'s increasing variables. Hence, after \$I\$'s increasing variables are raised, the sum of all dual variables related to \$J\$ is at least \$\frac{1 + \epsilon}{5 + 4\epsilon} p(I)\$. Since \$J\$ is still *uncovered* when \$I\$ is pushed on the stack, it must be

$$\frac{1}{5 + 4\epsilon} p(J) \geq \frac{1 + \epsilon}{5 + 4\epsilon} p(I)$$

which implies

$$p(J) \geq (1 + \epsilon)p(I). \quad \blacksquare$$

**Lemma 3** *Algorithm 1 takes*

$$O\left(\frac{T}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$$

*many communication rounds, where \$T\$ is the time required to compute a MIS in the conflict graph.*

**PROOF.** We first prove that each main phase of our algorithm, takes \$O(\frac{T}{\epsilon} \log \frac{P\_{\max}}{P\_{\min}})\$ many rounds. To do this we bound the number of independent sets computed in any phase of our algorithm. Let \$l\$ be an arbitrary such phase and let \$S\_{l\_1}, \dots, S\_{l\_k}\$ be the collection of maximal independent sets computed in phase \$l\$, indexed according to the order they are computed. From the fact that all such independent sets are maximal it follows that there is a set of instances \$I\_{l\_1}, \dots, I\_{l\_k}, I\_{l\_p} \in S\_{l\_p}\$ such that \$I\_{l\_p}\$ is incompatible with \$I\_{l\_{p+1}}\$.

From the previous claim, it follows that \$p(I\_{l\_k}) \geq (1 + \epsilon)^k p(I\_{l\_1})\$ which implies that in each phase \$O(\frac{1}{\epsilon} \log \frac{P\_{\max}}{P\_{\min}})\$ independent sets are computed.

We conclude the proof by noting that the number of phases is \$O(\log \frac{L\_{\max}}{L\_{\min}})\$. ■

We now give a bound on the length of each message, sent during the execution of the algorithm. At each step of our algorithm, processors need to compute a conflict graph over instances which are still uncovered at that step.

At the beginning, all instances are uncovered. Hence, in order to compute such conflict graph each processor sends to all his neighbors, the start-time and the end-time of the time window corresponding to his job, together with the length of his job. Let  $e_{\max}$  be the maximum end-time over all instances. It follows that, at the beginning, the length of each message is  $O(\log e_{\max})$ .

The value of the dual variables may change throughout the execution of the algorithm. As a consequence, the set of uncovered instances may change as well. To keep all processors updated, we proceed in this way.

Whenever a processor changes the value of some dual variables, he communicates the new value to all his neighbors. At each step, each processor changes the value of at most four variables and each of these values are bound by the maximum profit of an job. Thus, messages with length  $O(\log P_{\max})$  suffice.

In order to find a maximal independent set in the conflict graph we can use the randomized algorithm proposed by Luby [6]. Since time is discrete, the degree of each node in any conflict graph is finite. The number of communication rounds required to compute such an independent set is  $O(\log N)$ , where  $N$  denotes the size of the input. Alternatively, we can compute the independent sets by means of a network decomposition of the underlying communication graph of the processors. This can be done for instance in  $O(\log^2 n)$  many rounds, where  $n$  is the number of processors, using [5]. The details are omitted from this extended abstract.

This observation together with Lemma 1 and Lemma 3 imply the following theorem.

**Theorem 1** *For any given  $\epsilon > 0$ , there is a randomized  $\frac{1}{(20+\epsilon)}$ -approximation algorithm for the distributed scheduling problem which takes  $O\left(\frac{T}{\epsilon} \log \frac{L_{\max}}{L_{\min}} \log \frac{P_{\max}}{P_{\min}}\right)$  many rounds, where  $T$  is the time needed to compute a MIS in the conflict graph. The length of each message is  $O(\log e_{\max} + \log P_{\max})$ .*

By plugging in a different subroutine to compute a MIS in the conflict graph or, alternatively, a network decomposition of the communication graph we obtain the running times listed in the introduction.

## 6. REFERENCES

- [1] F. Chudak, T. Erlebach, A. Panconesi and M. Sozio. Primal-Dual Distributed Algorithms for Covering and Facility Location Problems *Manuscript*, 2006.
- [2] S. Khuller, U. Vishkin, and N. Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *J. Algorithms*, 17(2):280–289, 1994.
- [3] F. Kuhn and R. Wattenhofer. Constant-time distributed dominating set approximation. In *Proceedings, ACM Symposium on Principles of Distributed Computing*, pages 25–32, 2003.
- [4] F. Kuhn, T. Moscibroda, and R. Wattenhofer. On the Locality of Bounded Growth, *24th ACM Symposium on the Principles of Distributed Computing (PODC 05)*.
- [5] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(1993) pages 441–454.
- [6] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1053, 1986.
- [7] A. Panconesi and A. Srinivasan. The local nature of delta-coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.
- [8] A. Panconesi and A. Srinivasan. On the complexity of Distributed Network Decomposition. *Journal of Algorithms* 20, 356–374 (1996).
- [9] Fabrizio Grandoni, Jochen Könemann, Alessandro Panconesi and Mauro Sozio. A primal-dual bicriteria distributed algorithm for capacitated vertex cover. Accepted for publication in *SIAM Journal on Computing (SICOMP)*.
- [10] Amotz Bar-Noy and Reuven Bar-Yehuda and Ari Freund and Joseph Naor and Baruch Schieber. A unified approach to approximating resource allocation and scheduling, *Journal of the ACM*, 48,1069–1090,2001.
- [11] Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming, *STOC 1993*, pages 448–457.
- [12] Fabian Kuhn and Thomas Moscibroda and Roger Wattenhofer. The price of being near-sighted, *SODA 2006*, pages 980–989.