An LP-Free Approximation Algorithm for Scheduling on Unrelated Processor Systems

Teofilo F. Gonzalez
Department of Computer Science,
University of California,
Santa Barbara, CA, 93106
teo@cs.ucsb.edu

Abstract

The currently best approximation algorithm for non-preemptive scheduling on unrelated processor systems involves the solution of a set of linear programming problems. We show that each of these linear programming problems can be solved via a procedure that solves a set of problems via dynamic programming. Thus, by incorporating our procedures to with the previous approximation algorithm we have an LP-free (i.e., not requiring linear programming) approximation algorithm for non-preemptive scheduling on unrelated processor systems.

1 Introduction

We present LP-free (i.e., not requiring linear programming) approximation algorithms for non-preemptive scheduling on unrelated processor systems.

Deterministic scheduling problems have been studied for several decades by researchers from many different disciplines, including Operation Research, Applied Mathematics, Management Science, Electrical Engineering, and Computer Science and Engineering. The main reason behind the general interest in scheduling is that it has a wide range of applications in all sorts of industrial and commercial environments. The applications include (CPU) processor scheduling, instruction scheduling, machine shop scheduling, assembly line scheduling, etc. Because of all the research activity in deterministic scheduling, there is a wide range of techniques to generate suboptimal solutions to these problems, since the problem of generating an optimal solution to most of these problems is likely to be computationally intractable unless P = NP.

The basic deterministic scheduling problem consists of a set of n independent jobs to be executed by a set of m processors. Each job i has to be executed for p_i units of time by only one of the processors. A (nonpreemptive) schedule is an assignment of jobs to processors at time intervals in such a way that (a) each job is executed continuously for p_i units of time by one of the processors; and (b) each processor executes at most one job at each time unit. A schedule is said to *preemptive* if one allows parts of the same job to be executed by different processors at different time periods. In a *c-preemptive* schedule parts of the same job may be executed concurrently by two or more processors. The finish time of a schedule is the latest time at which a job is being executed. A schedule with least finish time amongst all feasible schedules is called an *optimal finish time* schedule.

In some applications the processors are built from different technologies, or some processors might not be completely dedicated to the processing of the jobs. This type of processors are called *uniform* because some processors are uniformly faster than the others. The speed at which a processor j executes a job is $s_j \geq 1$. Thus, job i takes p_i/s_j time when executed by processor j.

A more general situation arises when the processors have different capabilities and therefore are called *unrelated.* The time required by processor j to execute job i is $p_{i,j} > 0$, i.e., it depends not only on the processor executing the job, but it also depends on the type of job. For example, some processors have hard-wired float point operations, or a large number of registers, or large local memories. Thus different types of jobs would take different time to execute on these different type of processors without satisfying the uniformity criteria.

In this paper we present an LP-free approximation algorithm for scheduling a set of n jobs on m unrelated processors. Our algorithm generates schedules with finish time within two of the optimal finish time. There are several approximation algorithms for our problem as well as for restricted and generalized versions ([3], [2]). The approximation algorithm with the smallest approximation bound is the one developed by Lenstra, Shmoys, and Tardos ([3]) which invokes the solution of a set of linear programming problems. Our algorithm solves each of the linear programming problems by executing a set of dynamic programming procedures. The approximation bound is the same as the previous algorithm, but the time complexity is polynomial on n and m (rather than polynomial on n, m and the number of bits required to represent the input).

2 The Algorithm

The only approximation algorithm with a constant approximation bound for our scheduling problem was developed by Lenstra, Shmoys and Tardos [3]. The algorithm needs to solve a set of corresponding c-preemptive decision problems. These problems are formulated as linear programming problems as follows.

Is there a solution to

$\sum_{j=1}^{m} x_{i,j} = 1$ for $1 \leq i \leq n$

$\sum_{i=1}^{n} p_{i,j} x_{i,j} \leq d$ for $1 \leq j \leq m$

$0 \leq x_{i,j} \leq 1$ for all i, and j.

Each $p_{i,j}$ value is either less than or it is equal to d, or is equal to infinity. Assume all the $p_{i,j}$'s are real values. The other case can be solved similarly. In what follows we solve this decision problem.

Our algorithm is fairly simple. The first step it constructs an initial assignment x in which all jobs are assigned to one of the processors in which the job has the smallest workload, i.e., job i is assigned to processor j if $p_{i,j} = min\{p_{i,l} | 1 \leq l \leq m\}$.

For the assignment of jobs to processors x we define $tproc_j$ as $\sum_{i=1}^{n} x_{i,j} \cdot p_{i,j}$ for $1 \leq j \leq m$. Processor j is called *critical* in assignment x if $tproc_j > d$; it is called *tight* if $tproc_j = d$; otherwise we refer to it as an *idling* processor.

A *transformation path or t-path*, $TP(s,t)$, for assignment x from processor s to processor t is a sequence of $k-1 \geq 1$ jobs $i_1, i_2, \cdots, i_{k-1}$ and a sequence of $k \geq 2$ processors $j_1, j_2, \cdots, j_k$ such that $s = j_1$, $t = j_k$, and for $1 \leq l < k$ job i_l is assigned at least in part to processor j_l, i.e., $x_{i_l, j_l} > 0$. The value of $k - 2 \geq 0$ is called the number of *intermediate processors* in the t-path $TP(s,t)$. The *index* of t-path $TP(s,t)$ is the tuple $(i_1, i_2, \cdots, i_{k-1})$. Note that there may be many different t-paths between a pair of processors and the number of intermediate processors in these paths may also be different. Let A and B be two different t-paths with the same number of intermediate processors. We say that t-path A has a smaller index than t-path B if the index of A is lexicographically smaller than the index of B. A t-path, $TP(1,8)$, with $k = 6$; $j_1 = 1, j_2 = 5, j_3 = 3, j_4 = 7, j_5 = 6, j_6 = 8$; and $i_1 = 9, i_2 = 6, i_3 = 4, i_4 = 2, i_5 = 3$ is shown in Figure 1(a), and in Figure 1(b) we show its graphical representation.

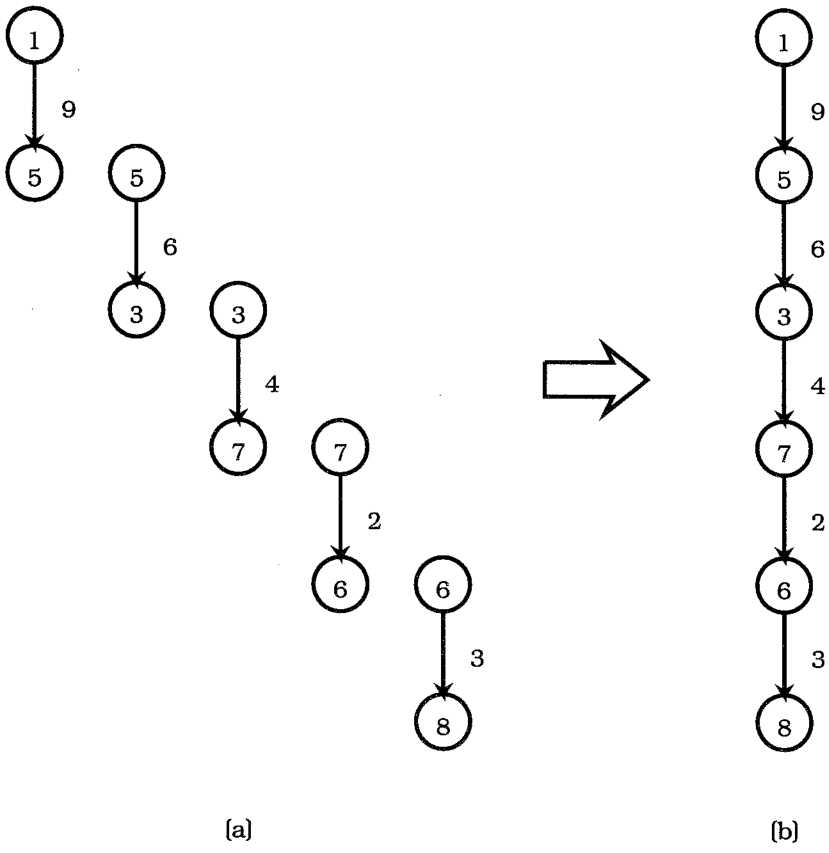

Figure 1: (a) T-path $TP(1,8)$. (b) Representation of the t-path $TP(1,8)$.

The t-path $TP(s,t)$ can transform c_1 units of workload from processor s to c_k units of workload on processor t if for $1 \leq l < k$ one can transfer c_l units of workload from job i_l assigned to processor j_l (i.e. if $c_l \leq x_{i_l,j_l} \cdot p_{i_l,j_l}$) to $c_{l+1} = c_l \cdot \frac{p_{i_l,j_{l+1}}}{p_{i_l,j_l}}$ units of workload for job i_l on processor j_{l+1}. In this case we say that the δ-factor along the *t-path* $TP(s,t)$ is equal to $\frac{c_k}{c_1}$. The portion of the t-path $TP(s,t)$ that begins at processor j_p and ends at processor j_q for any $1 \leq p < q \leq k$ has δ-factor equal to $\frac{c_q}{c_p}$. Therefore, for $p < l < q$ the δ-factor from p to l along $TP(s,t)$ times the δ-factor from l to q along $TP(s,t)$ is equal to the δ-factor from p to q along $TP(s,t)$, i.e., $\frac{c_l}{c_p} \cdot \frac{c_l}{c_q} = \frac{c_q}{c_p}$.

The maximum workload which can be transferred out of processor s along the t-path $TP(s,t)$ is called the *capacity* of the t-path $TP(s,t)$. We define $\delta(s,t)$ as the smallest δ-factor of any t-path from processor s to processor t, and we define *cap*(s,t) as the capacity of a t-path $TP(s,t)$ with δ-factor equal to $\delta(s,t)$. An assignment x is said to be *irreducible* if all the t-cycles (i.e., a t-path that begins and ends in the same processor, $TP(s,s)$), and all the t-paths, $TP(s,t)$ that begin at a critical processor s have a $\delta-$factor greater than or equal than 1. At each iteration we construct an irreducible assignment. This assignment has either a larger number of tasks assigned 100The algorithm terminates when there are no critical processors, or no idling processors. The answer to the decision problem is yes when an assignment in which none of the processors are critical is generated, and it terminates with the answer no when there is at least one critical processor and there are no idling processors.

We define $\delta(s,t,k)$ as the smallest δ-factor of any $TP(s,t)$ t-path that goes through at most k intermediate processors. It is simple to see that for irreducible assignments $\delta(s,t) = \delta(s,t,m-1)$.

The following recurrence relation establishes a method for computing $\delta(s,t)$ for irreducible assignments.

$$\delta(s,t,l) = \begin{cases} min\{\frac{p_{i,t}}{p_{i,s}} | x_{i,j} > 0\} & l = 0 \\ min\{\delta(s,k,0) \cdot \delta(k,t,l-1) | 1 \leq k \leq m\} & l \geq 1 \end{cases}$$

By using standard dynamic programming techniques it is simple to show that $\delta(s,t)$ for all s and t can be computed in $O(m^3 + n \cdot m^2)$.

Our algorithm proceeds as follows from the irreducible assignment x. Using the procedure described above we compute $\delta(s,t)$ for all s and t from x, and label the processors critical, tight, or idling following our previous definitions. Suppose that there is at least one critical and one idling processor, as otherwise the algorithm terminates. For each critical processor i we define $S(i)$ as the set of shortest t-paths (with respect to the number of intermediate processors) from processor i to an idling processor whose δ-factor is minimum among all t-paths from processor i to an idling processor, i.e., the δ-factor of the t-path is equal to $min\{\delta(i,j)|$ processor j is idling$\}$. We shall refer to the t-path in $S(i)$ least index as the t-path $TP(i)$.

The union of all the t-paths $TP(i)$'s beginning on each critical processor forms a directed forest of reversed trees (children point to their parents) as shown in Figure 2, since each of these t-paths has least index among the shortest paths with least δ-factor. There is exactly one node in the forest for each critical processor,(represented by a circle), and there is at most one node for each idling processor (represented by a square) and each tight processor (represented by a triangle). Later on we show in our forest the leaves are idling processors, the roots are critical processors, and all other nodes (i.e., representing the intermediate processors) are critical or tight processors. Let i and j be two processors present in the forest such that i is a predecessor of j in the forest. Then the δ-factor of the portion of the path from i to j in the forest is equal to $\delta(i,j)$.

A t-path is said to be a *direct* t-path if all the intermediate processors it goes through (if any) are tight processors, otherwise the t-path is said to be *indirect.* Now we take any direct t-path $TP(s,t)$ in S originating at a critical processor s and ending at an idling processor. We apply the t-path $TP(s,t)$ until either processor s becomes tight, processor t becomes tight, or we have transfered workload equivalent to the capacity of the t-path. As a result of this operation we generate another assignment x. This process is repeated until an assignment in which there are no critical processors, or there are no idling processors. Later on we establish that each of the assignments our algorithm generate is irreducible which later on will be used to establish optimality.

Suppose now that δ units of workload are transferred along the t-path $TP(s)$ of each critical processor s. Then, the workload on each processor j is changed by $\delta \cdot q_j$, for some real value q_j. Note that $q_j \geq 1$ for each non-critical processor j, and $q_j = -1$ for each critical processor j. In addition the workload for job i on processor j will change by the amount $\delta \cdot r_{i,j}$, for

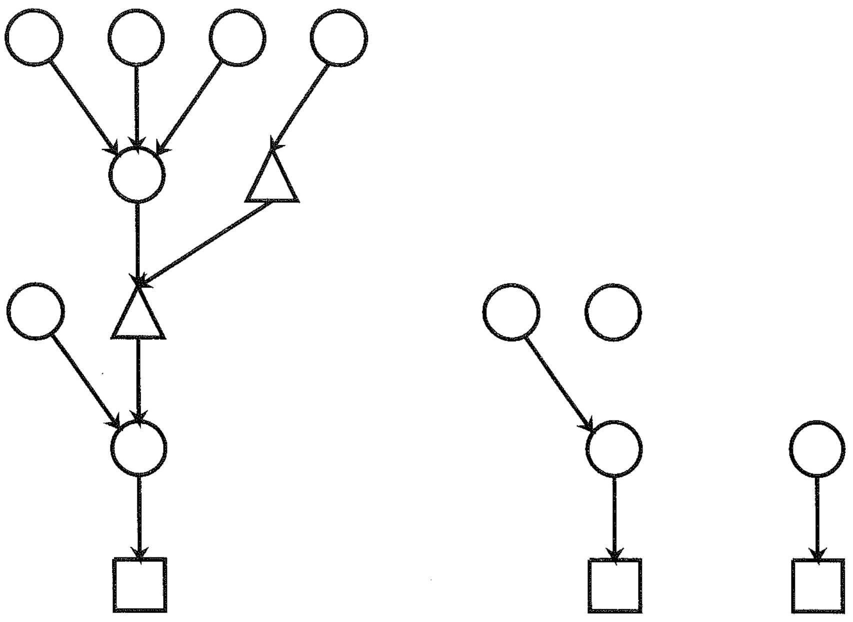

Figure 2: Forest of t-paths.

some real value $r_{i,j}$. Define

$$\Delta = max\ \{\delta | \delta \cdot r_{i,j} \leq x_{i,j} \cdot p_{i,j},\ \text{and}\ t_{max} - \delta \geq tproc_j + \delta \cdot q_j\},$$

i.e., Δ is the maximum amount one can transfer along the t-paths $TP(s)$ of the critical processors so that at least one previous non-zero $x_{i,j}$ becomes zero in the resulting assignment and/or at least one non-critical processor becomes critical in the resulting assignment.

```
procedure LP-FREE($p, n, m$);
  Construct the initial assignment $x$ in which all jobs are assigned to one
    of the processors in which the job has the smallest workload, i.e., job $i$
    is assigned to processor $j$ if $p_{i,j} = min\{p_{i,l} | 1 \leq l \leq m\}$;
  while there is at least one non-critical processor for assignment $x$ do
    Find a set of t-paths that decrease the workload of all
      critical processors uniformly with a minimum total workload increase
      in the non-critical processors without overloading them;
    Apply all the t-paths concurrently;
```

```
      Let $x$ be the new assignment;
    endwhile
  end of procedure LP-FREE;
```

We claim that at each iteration the assignment x is irreducible. In the following lemma we establish our invariant.

Lemma 2.1 *At each iteration of algorithm $LP - FREE$ the assignment x is irreducible.*

Proof: The lemma obviously holds for the first assignment x, since each job is assigned completely to one of the processors in which the job has the smallest workload, i.e., job i is assigned to processor j if $p_{i,j} = min\{p_{i,l} | 1 \leq l \leq m\}$. We know show that if the conditions of the lemma hold at the beginning of the iteration, then they also holds at the end of the iteration. We shall refer to the assignment x at the end of the iteration as the new assignment x.

We claim that each the t-paths $TP(s)$ selected by the algorithm goes through only critical processors. Suppose not. Suppose the path $TP(s)$ that starts at processors s and ends at processor t also goes through non-critical processors. Let processor y be the first non-critical processor that the $TP(s)$ goes through. Since assignment x is irreducible, we know that the portion of the t-path $TP(s)$ that starts at the non-critical processor y and ends at the non critical processor t has a δ-factor greater than or equal to one. Therefore, the δ-factor of the portion of the t-path $TP(s)$ that starts at s and ends at y is not-larger than the one of t-path $TP(s)$ and the number of intermediate processors in that path is smaller than those in the t-path $TP(s)$. But this contradicts the way the algorithm selects the t-path $TP(s)$. Hence, each the t-paths $TP(s)$ selected by the algorithm goes through only critical processors.

Now we claim that in the new assignment all the t-paths that end at a non-critical processor have a δ-factor greater than or equal to 1. We prove this by contradiction. Suppose not. Suppose that in the new assignment there is a t-path without intermediate processors that starts at processor s, ends at the non-critical processor t, and has a δ-factor less than one. Let TP be a t-path with least δ-factor that ends at a non-critical processor. Let $j_1, j_2, ..., j_k$ be the processor indices in t-path TP, and let $i_1, i_2, \ldots i_{k-1}$ be the job indices associated with this path. Since the t-path TP has the smallest

δ-factor, it must be that for $1 \leq p < q \leq m$ the δ-factor of the portion of the path from processor j_p to processor j_q is equal to $\delta(p,q)$. Clearly, the t-path TP did not exist in the previous x assignment because it was an irreducible assignment. so it must be that the t-forest made non-zero the value of at least on x_{i_l,j_l} for some $1 \leq l \leq k-1$. Assume without loss of generality (as otherwise we could compress the t-path TP) that that x_{i_l,j_l} for all $1 \leq l \leq k-1$ were increased from zero to a positive value when we applied the last t-forest. We now show that $\delta(i_1, i_k) \geq 1$, which contradicts the previous assumption.

In what follows we use $\delta_o(i,j)$ to refer to $\delta(i,j)$ at the previous iteration. Since the previous x assignment was irreducible, we know that $\delta_o(j_1, root(j_1) \geq 1$. Lets show that $\delta(j_1, j_2) \geq \frac{1}{\delta_o j_2, root(j_2)}$

□

To prove correctness we need to establish inductively that each of the x assignments constructed by the algorithm is irreducible. Then we show that a schedule with a finish time smaller than the one generated by our algorithm implies that the last x assignment is reducible. Which contracts our earlier proof. Thus we have the following result.

Theorem 2.1 *Our algorithm generate minimum finish time schedules for the c-preemptive scheduling problem on unrelated processor systems.*

Proof: For brevity we omit the proof.

□

We performed an experimental evaluation of our algorithm and the simplex method. For brevity we cannot present all the results, but we should mention that on problems with 120 jobs and 60 processors our algorithm takes about 1.3 CPU hours, whereas the simplex method takes about 5 CPU hours. The experiments were carried out on a SUN LX machine.

As mentioned before, algorithm can be easily adapted to the case when some of the $p'_{i,j}s$ have the value of infinity. Thus, our algorithm can solve the linear programming problems in Lenstra, *et. al.* algorithm. Thus we have an LP-free approximation algorithm for scheduling on unrelated processor systems. The approximation bound is the same as before, but the time complexity bound in polynomial on n and m, rather than polynomial on n, m, and the number of bits that represent the input.

3 Discussion

It is intersting to note that it can be generalized to provide an LP-free algorithm for preemptive scheduling independent jobs on unrelated processo systems.

References

[1] T. F. Gonzalez, E. L. Lawler, and S. Sahni, "Optimal Preemptive Scheduling of Two Unrelated Processors," *ORSA Journal on Computing*, 2(3), pp. 219 – 224, 1990.

[2] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, E. Tardos, "Sequencing and Scheduling: Algorithms and Complexity," BS-R8909, Center for Mathematics and Computer Science, 1989.

[3] J. K. Lenstra, D. B Shmoys and E. Tardos, "Approximation Algorithm for Scheduling Unrelated Parallel Machines," *Mathematical Programming*, 46, (1990), pp. 259 – 271.